NTNU – Trondheim
Norwegian University of
Science and Technology

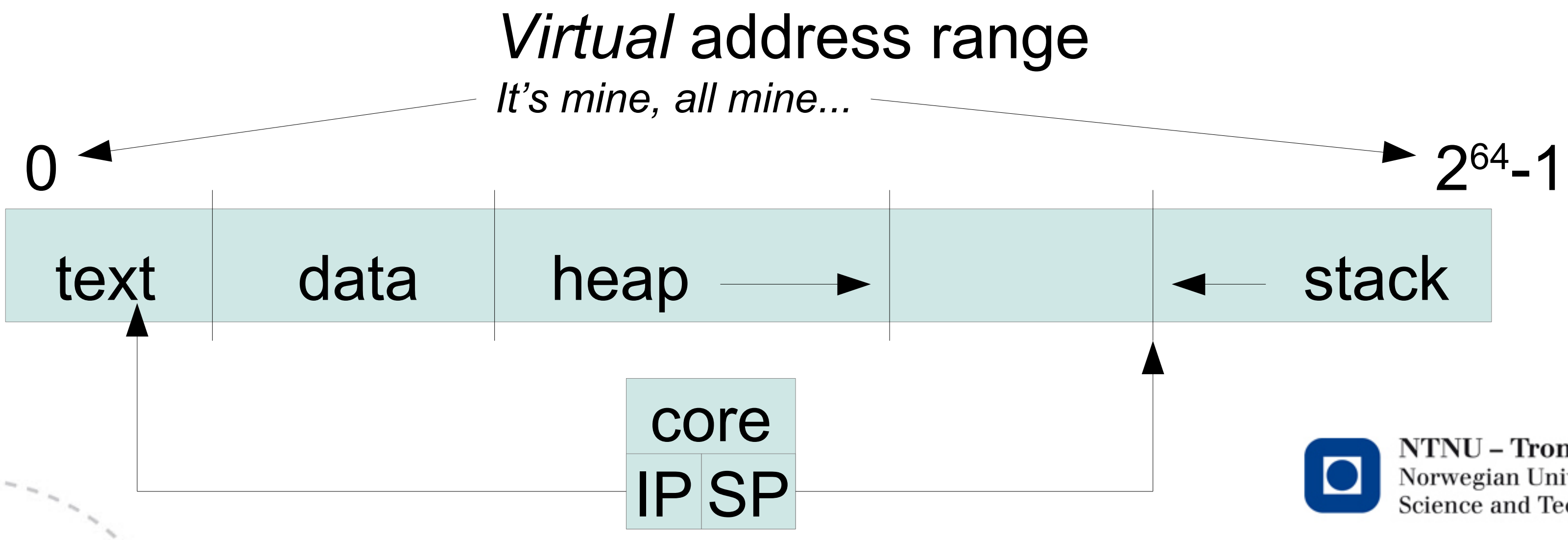
Introduction to MPI

Jan C. Meyer

TDT4200, 17.09.2018

A process and its thread

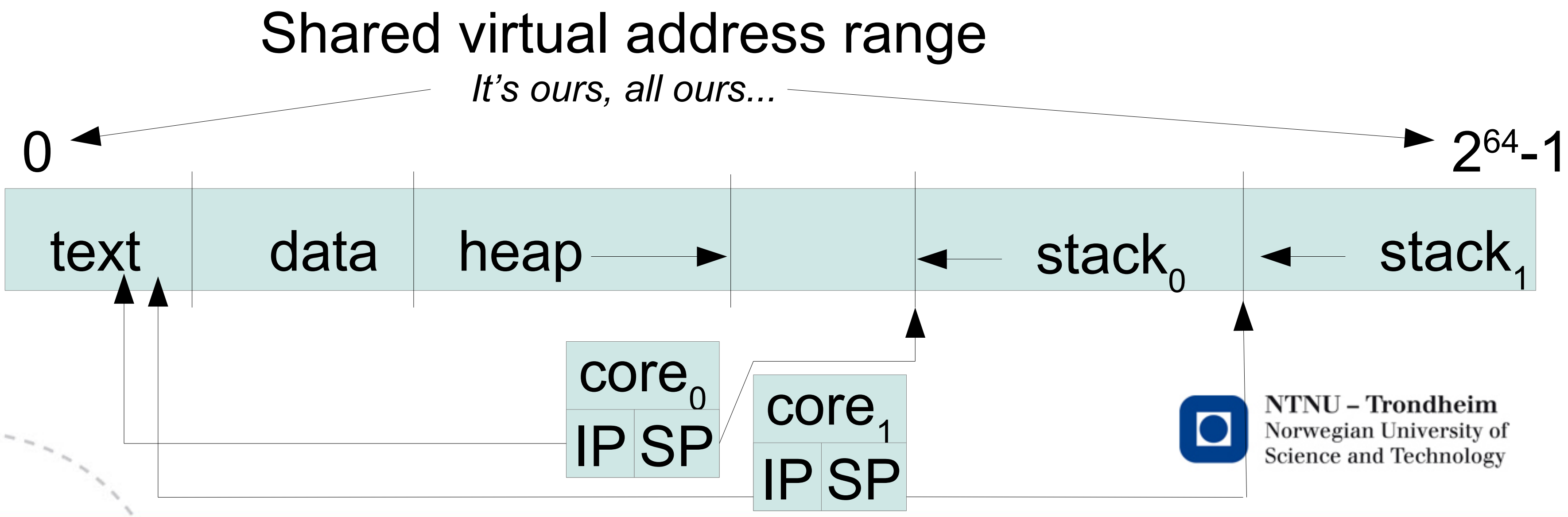
- Below is a (simplified) sketch of what a single CPU core deals with when running a sequential program
  - Instruction Pointer (IP) identifies next operation to carry out
  - Stack Pointer (SP) says where to find the local values of the presently running function
  - No other program exists



NTNU – Trondheim
Norwegian University of Science and Technology

A process and two threads

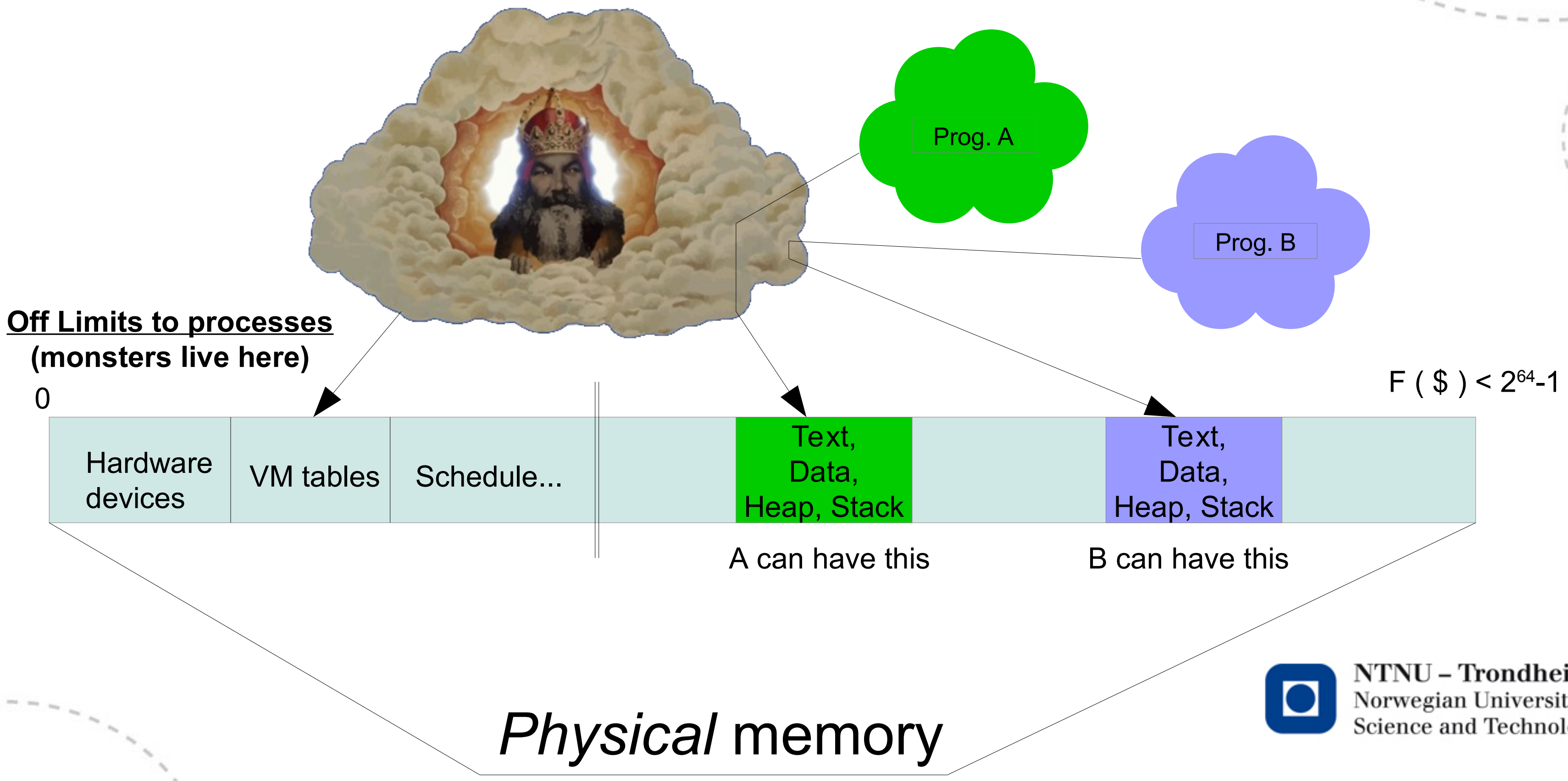
- By doubling program positions and local workspaces, two cores can run two different functions simultaneously
  - They can share all the contents of data/heap segments (global values, open files, dynamically sized arrays, *etc.*) if they're careful
  - Still no idea that there are other programs on this computer



NTNU – Trondheim
Norwegian University of Science and Technology

Enter: The Operating System

- Assigns resources to program contexts by need
- Activates hardware on request
- Protects programs from each other (and themselves)



The
Message Passing Interface

- You can think of MPI as a delivery service between processes; a process can
  - Send a message, labeling it with size, destination, layout, contents
  - Receive a waiting message, and place contents in its own memory
- For this to happen, MPI must attach to processes when they launch, so that it can direct their messages through the underlying machinery
  - You can launch 4 (simultaneous) instances of your program like this:

    ```
    ./myprogram & ./myprogram & ./myprogram & ./myprogram
    ```

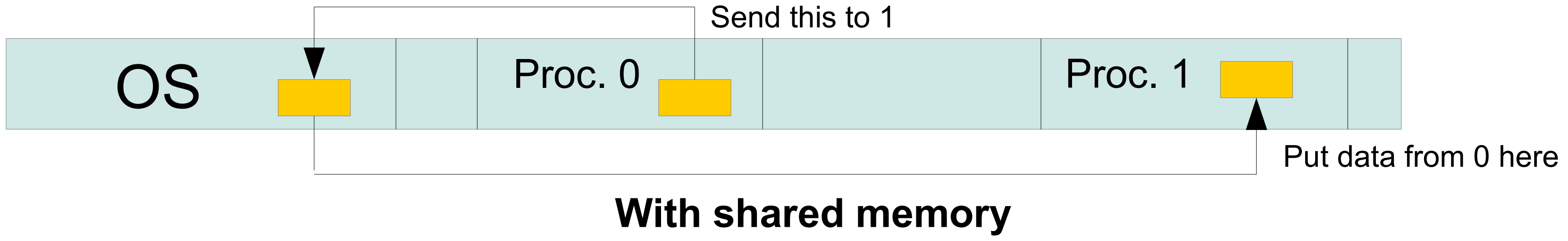
  - If you install MPI and do it like this:

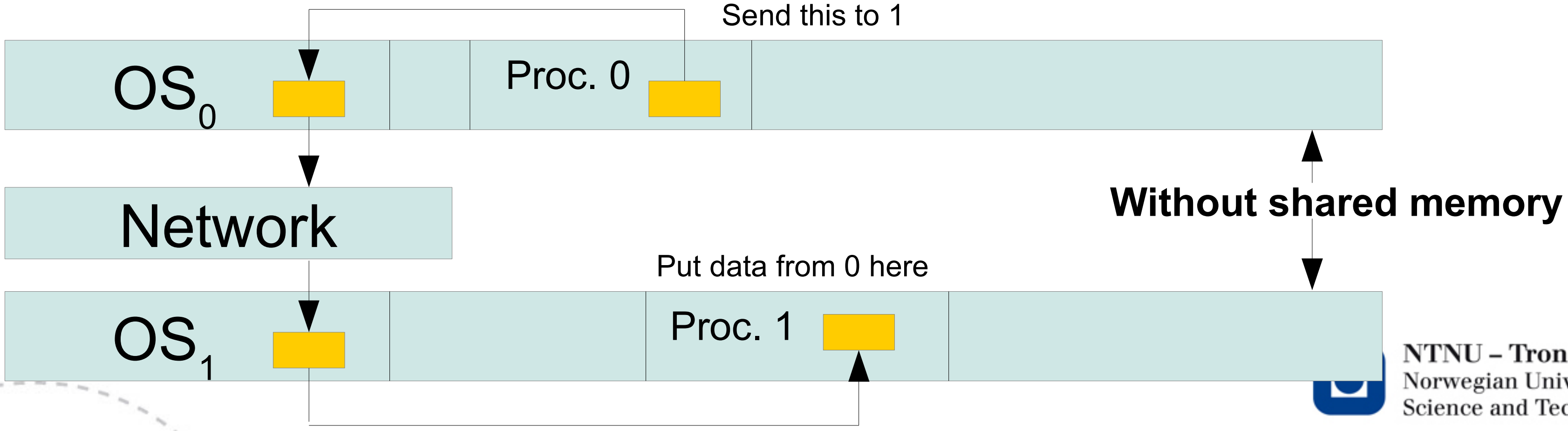
    ```
    mpirun -np 4 ./myprogram
    ```

    pretty much the same thing happens, except that MPI can track the instances and move data between them

What’s great about it

- MPI works out whether your processes are on the same computer, program doesn’t have to care:





NTNU – Trondheim
Norwegian University of Science and Technology

Shared, distributed, who cares?

- If you were looking for a computer at around NOK60.000.000 in 2012, these were two options:

UltraViolet 2000
4096 cores, shared memory

Altix ICE 4800
22464 cores, distributed memory

(shared memory cost grows very quickly)

What’s not so great about it

- When 200 processes need the same data, they need 200 copies of it
  - (and must manage its consistency themselves)
- Programs that are full of sending and receiving operations grow a bit longer and more complicated than programs where data moves transparently
  - (conversely, this also makes the cost of data movement visible in the source code, so you can change how it’s done)

Footnote: *Hybrid* programs

- Separate computers *(nodes)* these days are small multi-core/shared-memory platforms
- The most resource-economic way to program them is to use a mixture of threads and processes
- Today, we're just talking about MPI, though

Launching processes

- As mentioned,

  ```
  mpirun -np 16 ./program
  ```

  kicks off 16 separate copies of ./program
- They're all copies of the same code, so unless they get in touch with each other, they'll just do the same thing 16 times over.
- To get them talking, MPI must first set up its internal mechanisms, and when they're done, those have to be taken down again.
  - The functions `MPI_Init` and `MPI_Finalize` do exactly this.

NTNU – Trondheim
Norwegian University of
Science and Technology

Start and stop

- Here’s a minimal MPI-enabled program:

```
#include <stdio.h>
#include <mpi.h>

int
main ( int argc, char **argv )
{
    MPI_Init ( &argc, &argv );
    printf ( "Hello, world!\n" );
    MPI_Finalize();
}
```

- Compile, and run 4 copies:

```
% mpicc -o hello_seq hello_seq.c
% mpirun -np 4 ./hello_seq
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

Wrapper script for the compiler (for C++, use mpicxx or mpic++)

All processes still just do the same thing, though...

NTNU – Trondheim
Norwegian University of Science and Technology

SPMD programs

- **S**ingle **P**rogram **M**ultiple **D**ata is a name for the idea that if your code decides what to do by inspecting some data values, many copies of the Single Program will act differently if you give them Multiple Data

  (In terms of Flynn's taxonomy (Chapter 2.3), it's a way to merge MIMD work into one source code)

- MPI is full of functions where the same call will return different results on different processes
- The simplest one is MPI_Comm_rank, which just gives a unique number to each process

Hi, everybody!

- Here’s a (slightly) more interesting variant:

```
#include <stdio.h>
#include <mpi.h>

int
main ( int argc, char **argv )
{
    int rank, size;
    MPI_Init ( &argc, &argv );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    MPI_Comm_size ( MPI_COMM_WORLD, &size );
    printf ( "Hello from process %d out of %d\n", rank, size );
    MPI_Finalize();
}
```

- Try 4 copies again:

```
% mpicc -o hello_spmd hello_spmd.c
% mpirun -np 4 ./hello_spmd
Hello from process 0 out of 4
Hello from process 2 out of 4
Hello from process 1 out of 4
Hello from process 3 out of 4
```

Communicators

- A *communicator* (MPI_Comm) is a collection of processes that you want to group together
- A process can be a member of many, inside each it will have its own *rank* value, to separate it from the others
- The *size* of an MPI_Comm is its count of members
- When you call MPI_Init, it sets up one called MPI_COMM_WORLD that contains *all* your processes

  (so you can construct other communicators by selecting from that)
  - The size of MPI_COMM_WORLD is the `-np P#` number you give mpirun
  - Ranks are just assigned as the numbers from 0 to P-1
- There's a lot to say about communicators, but we'll just use this one today.

Sending and receiving

- A message features
  - Some data (a pointer to a place)
  - A number of elements (of some type)
  - A type of elements (message size = elements x type size)
  - Source and destination ranks
  - An arbitrary integer that you can tag it with for your own purposes
  - The communicator where the source and destination ranks are
  - An MPI_Status that contains what MPI thinks of the situation
- This is pretty much the same list at both ends of the transmission

NTNU – Trondheim
Norwegian University of
Science and Technology

MPI_Send, MPI_Recv

- These are the function prototypes for point-to-point messages:

```
MPI_Send (
    void *buffer,
    int elements,
    MPI_Type type,
    int destination,
    int tag,
    MPI_Comm communicator
);
```

```
MPI_Recv (
    void *buffer,
    int elements,
    MPI_Type type,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status status
);
```

The argument lists are long, but notice how similar they are...

NTNU – Trondheim
Norwegian University of
Science and Technology

Argument clinic

In short order,

1) *buffer* points to the transmitted data on the sending side, on the receiving side it points to a block of free memory where it will fit. You are in charge of these, MPI won't allocate anything.

2,3) Element count is just a number, the MPI_Type can be MPI_INT, MPI_DOUBLE, MPI_BYTE, … there's a table on p. 89.

4) The sender fills in where the message is going, the receiver where it should be expected from, these are rank values from the communicator

5) The *tag* is just a number you make up, but it has to be the same at sender and receiver. It's used for matching up the right pairs when you have several send/receive pairs going at the same time. We won't do that here.

6) We already discussed the communicator

(7) *status* is only relevant at the receiving end, it contains information that MPI figures out at run time, *e.g.* who was the sender if you post a "receive-from-anywhere" and such things. If you don't care, you can put MPI_STATUS_IGNORE.

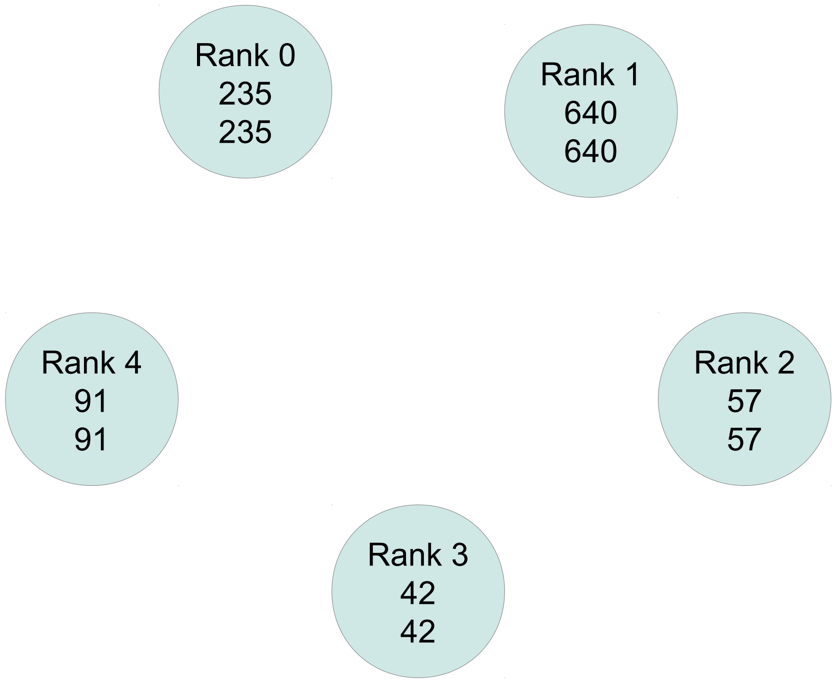
Passing values in a ring

(“duck duck goose” in MPI)

- As a practical exercise, we can pass values around a circle
- There’s a limit to how much code one can fit onto a slide like this, find implementation in ‘ring_maximum.c’
- I’ll illustrate what it does instead

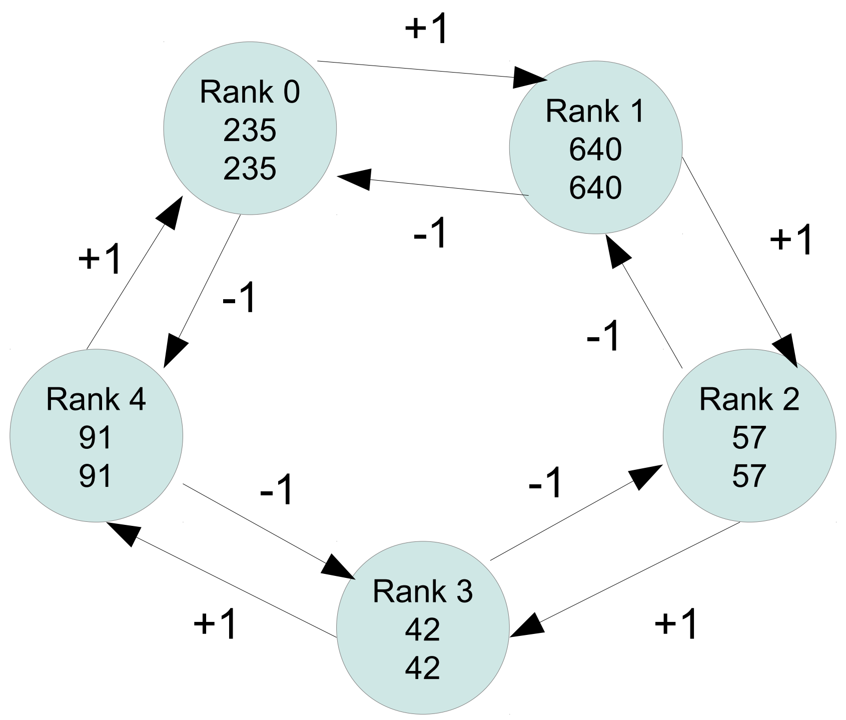
Starting out

- Suppose we start 5 processes, give them all an arbitrary number, and say that it's the biggest they have seen so far:



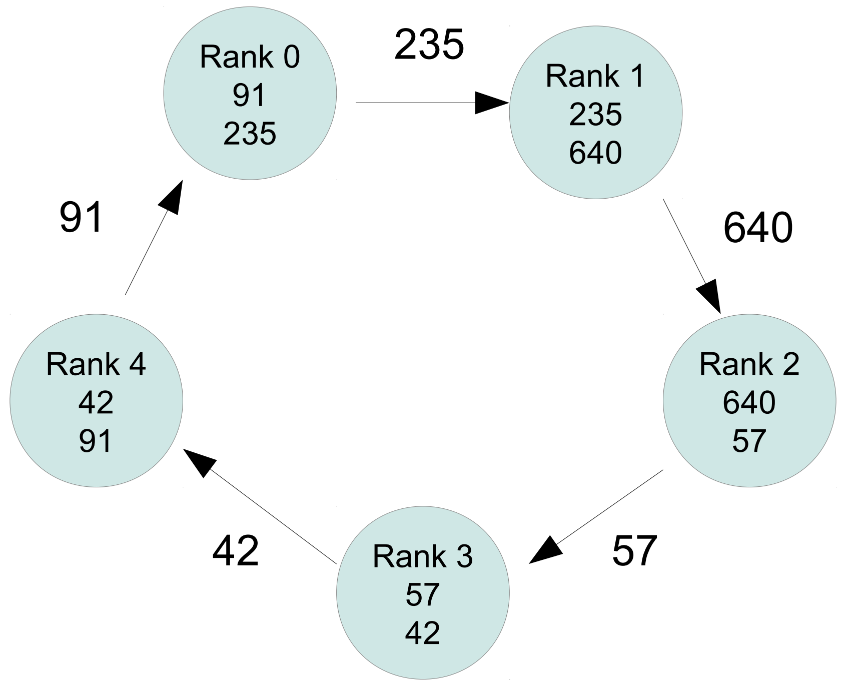
Find your neighbors

- (rank+1) modulo size is *right*, (rank-1) modulo size is *left*
- Add ‘size’ to rank-1, so that rank 0 gets 4 instead of -1 to the left



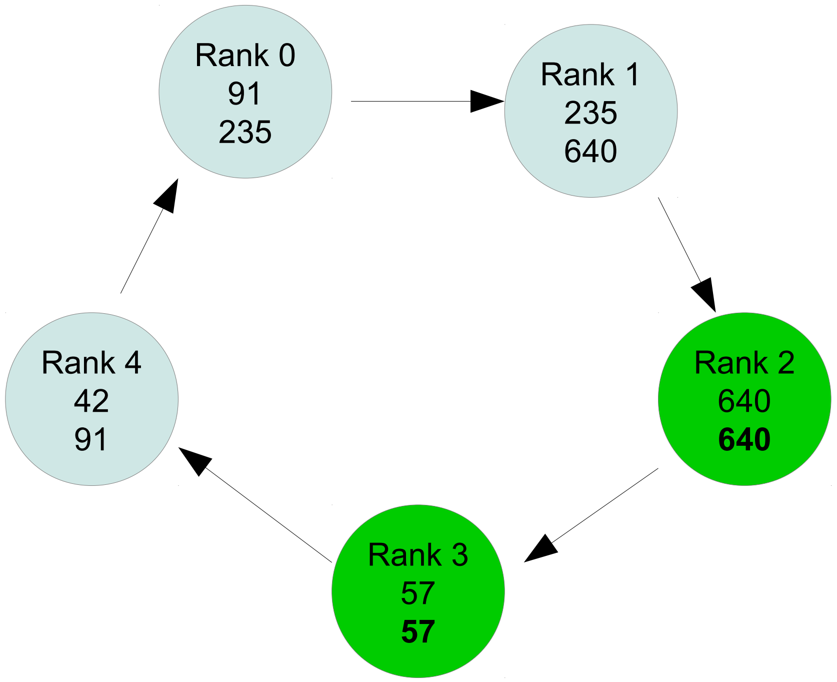
Make a step

- Send your value right, receive one from left:



Compare and update

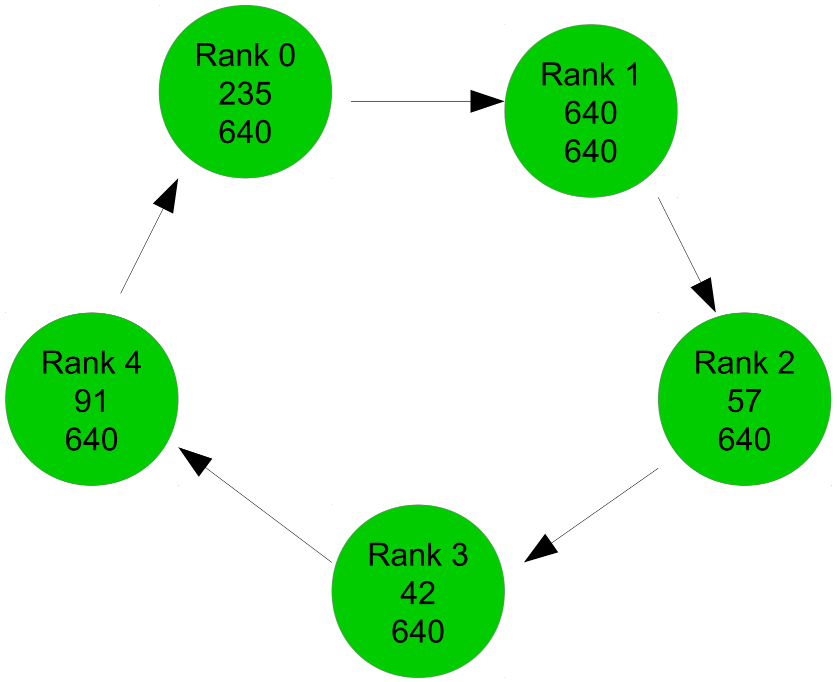
- If new value is greater than biggest seen, keep it



NTNU – Trondheim
Norwegian University of
Science and Technology

Make a step

- If we repeat this 5 times, all the values come back where they started, but every rank has seen every number passing by



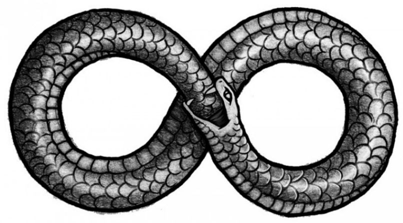
That’s all, in a way

- The MPI standard specifies hundreds of function calls for various purposes, it can look a bit daunting
- The whole thing can be implemented in terms of the 6 functions we’ve seen: Init, Finalize, Size, Rank, Send and Recv
- What we just did was to hand-implement a *global reduction*
  - We hard-coded a distributed array of P values, but the local value could easily be replaced with a section of a huge array
  - If you think it over, you’ll see the same pattern could also be used for the minimum, sum, product, and bitwise combinations of distributed arrays

Collective work

- Reductions are a subset of MPI's *collective operations*
- Collectives are function calls where every member of a communicator must call them and participate before they can finish
- Even though we built it with point-to-point messages, our step-and-repeat pattern is like that, it grinds to a halt unless every rank takes part in every iteration
- Collective communication is our next topic

Wait a minute...



- Our ring-method is not the Algorithm Of The Year, it has a serious problem
- You may have spotted it already... consider the following dance instructions:
  - Everyone stand in a circle
  - First, offer one hand to your right neighbor
  - Take the hand of your left neighbor after your right has accepted
- When followed to the letter, this could take a while.

  The program still works, what is going on?

NTNU – Trondheim
Norwegian University of
Science and Technology

Under the hood

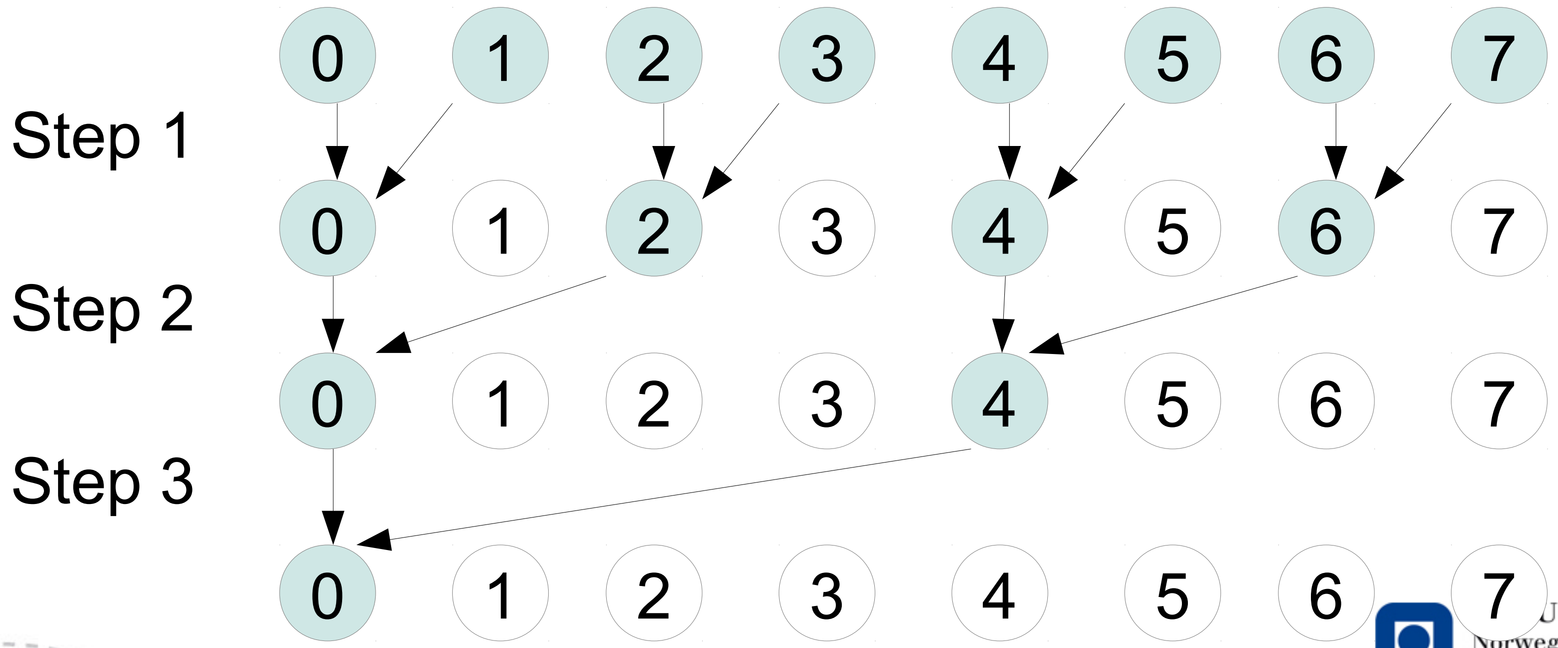
- MPI_Send is an implementation-defined default sending method with a few liberties
- Most implementations have a fixed internal buffer to
  - Whisk “small enough” messages and return control to the program before the sending is actually completed
  - Our variant where everyone sends before receiving will deadlock when the messages pass this implementation-defined threshold
- You can take explicit control of this
  - MPI_Ssend (‘synchronized send’) forces the program to wait until the transfer is complete, it has the exact same argument list as MPI_Send
  - If you swap Send for Ssend in the example program, it will always deadlock, exposing this issue

Painless pairwise messages

- The manual solution to break the cycle is to make a conditional, and let at least one rank do receive-before-send instead of send-before receive
- This is a hassle, so there is a dedicated call for doing one send and one receive at the same time, and MPI has to guarantee that it doesn't deadlock
- It's called MPI_Sendrecv, see chapter 3.7

Back to the collectives

- Instead of a ring, we could also organize a reduction by sending messages in a tree structure:



Tree vs. ring

- The ring takes P stages, the tree takes log(P)
- The ring gives every rank the reduced value, tree only collects it at the *root*
- In the ring, all communication is between neighboring ranks (physically close in the machine), the tree has an initial stage with neighbor-pairs and a final stage where a message travels across P/2 ranks

Which is better?

- The answer depends on network topology
  - If your processors are in fact connected in a ring, passing messages in a ring works very well
  - If they are connected in a tree of switches, the tree pattern doesn't incur the kind of cost it would on a ring-shaped network
  - If they are connected in a mesh, you can make clever rank placements to affect the cost
  - If they are connected in a hypercube…
  - You Get The Picture

Transparent pattern choices

- Network shapes differ from platform to platform
- A bad solution would be to program all the alternatives you can think of, read about the topology of your cluster, and put in the most suitable one yourself
- The MPI way is to provide function calls like
  - `MPI_Reduce ( input, output, count, type, operation, root, communicator )`
  - `MPI_Allreduce ( input, output, count, type, operation, communicator )`

  and let the platform vendor provide an MPI library that does what's best on the network you have

Reduce and Allreduce

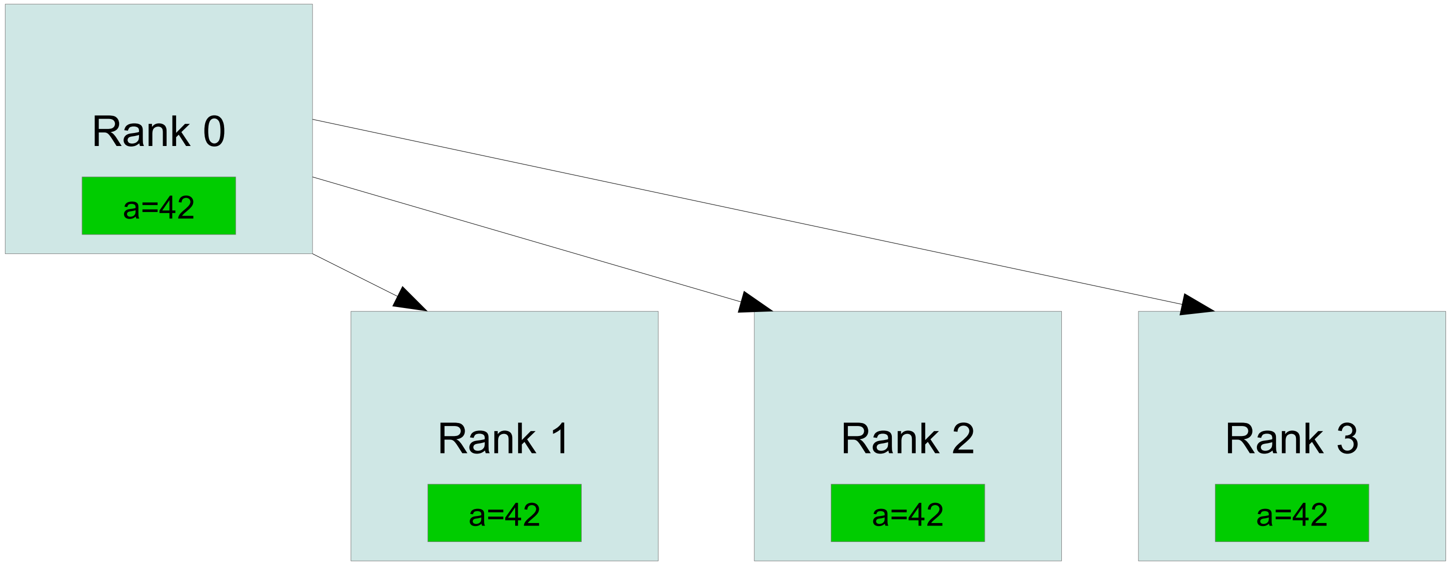
- The “operation” argument is a choice from min, max, sum, product, *etc. etc.*, see table on p. 104
- The “root” argument in Reduce is the rank where a single-destination collective (like our tree) collects the result
- There’s no “root” argument in Allreduce, because everyone gets a copy of the final result, like we did with the ring method

NTNU – Trondheim
Norwegian University of Science and Technology

Other collectives:

Broadcasting

- MPI_Bcast makes sure that everyone gets a copy of some data from a root rank (I'll use 0 as root):

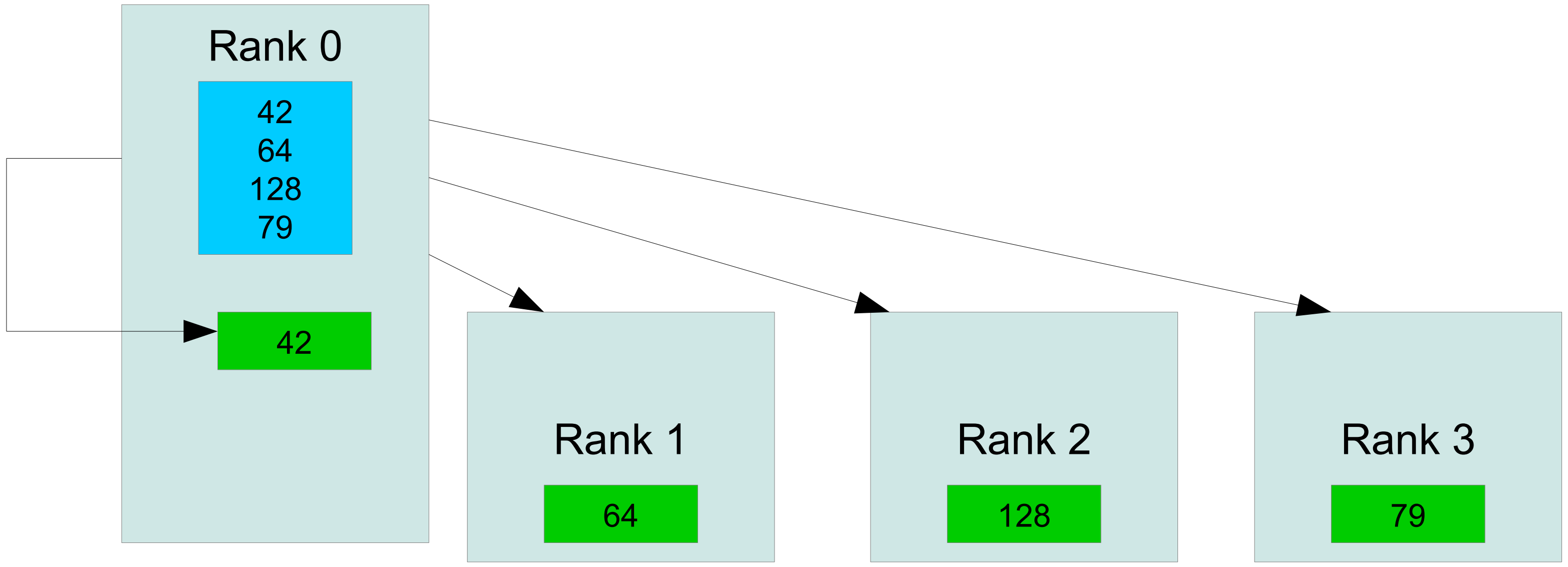


NTNU – Trondheim
Norwegian University of Science and Technology

Other collectives:

Scattering

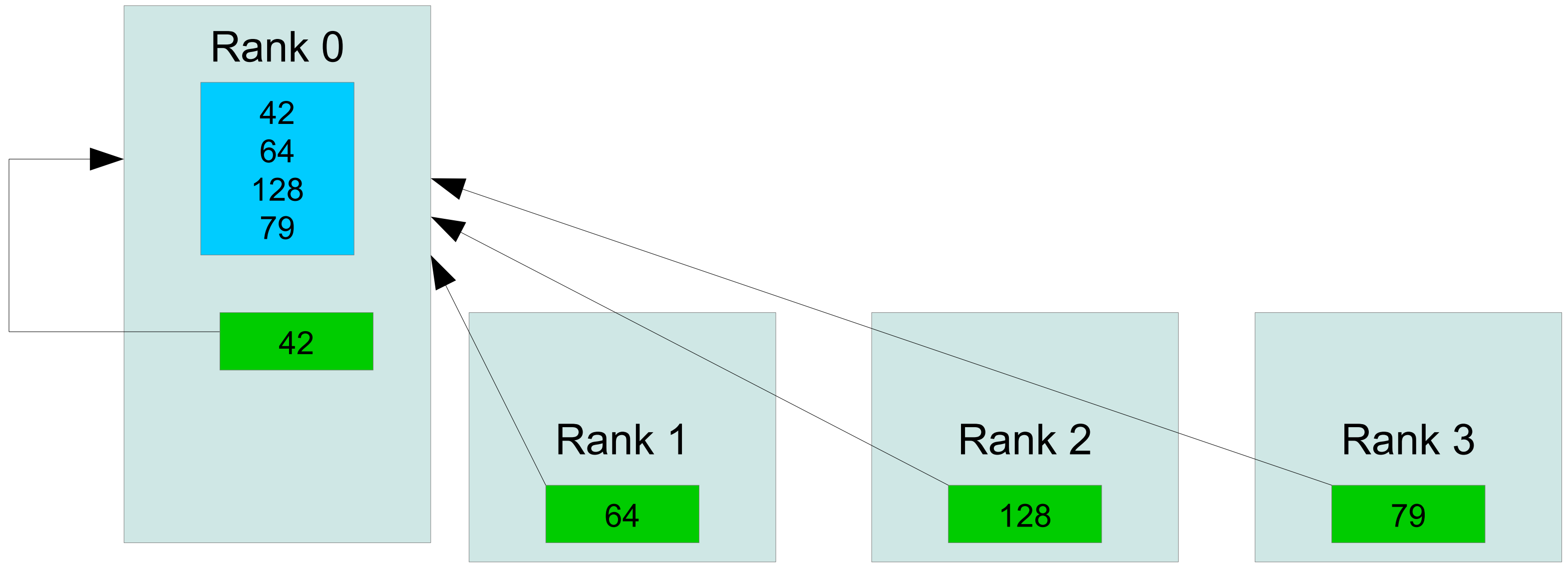
- MPI_Scatter takes a large lump of data at the root and distributes equal parts to everyone



Other collectives:

Gathering

- MPI_Gather collects equal parts from everyone and makes a big lump at the root



Global gather

- As with reductions, there is also a gather operation where every contributor ends up with a copy of the big lump at the end
- Unsurprisingly, it's called MPI_Allgather
- We don't have time to go through every argument list of every call here, but you shouldn't have to learn them by heart under any circumstances – they are thoroughly documented

Putting it all to use

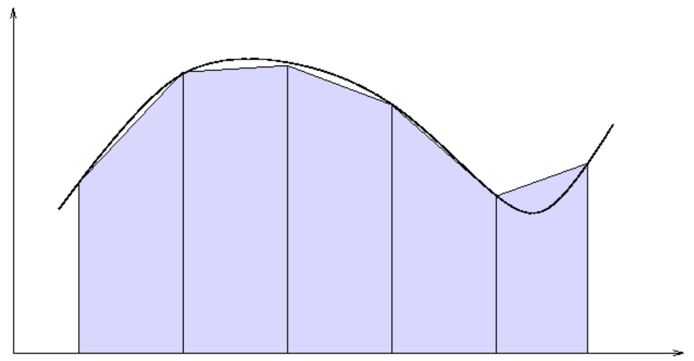
- We skipped past chapter 3.2, which is a simple numerical integration scheme in MPI
  - I don't think that example is very exciting, it just finds the area under a curve
  - I don't think it's very realistic either, because it doesn't handle any noticeable amount of data
  - It's pretty simple to read, so you can do that without my narration
- Instead, I've cooked up an example of my own that uses a similar numerical integration scheme, but also does something.
- I'll try to point out where to spot the similarity.

What the book does: Trapezoid rule

- Approximate a curve as a straight line between $f(x_0)$ and $f(x_1)$, that is,
- add up areas of

  $(x_1-x_0) * (f(x_0) + f(x_1)) / 2$

  for a whole lot of x-s (that we don't really need to keep)



FDTD integration
(Finite Difference Time Domain)

- Take a bunch of values that represent something in space (just a line, for simplicity)

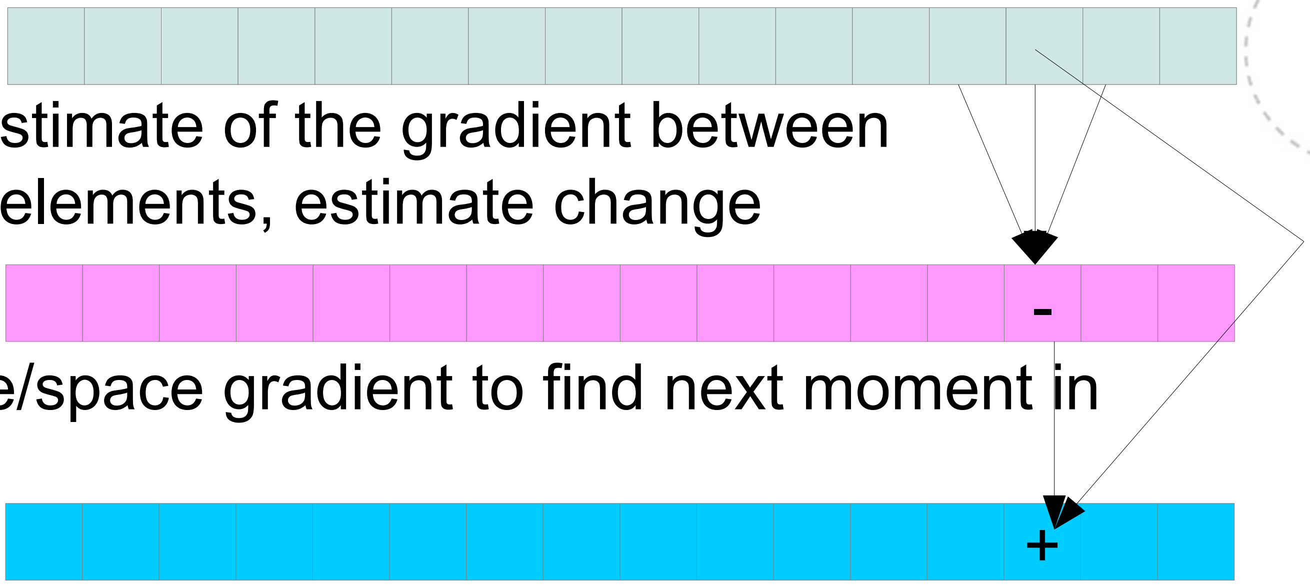
  time=0

- Find some estimate of the gradient between neighboring elements, estimate change

  dx/dt:

- Add this time/space gradient to find next moment in time

  time=1:



Similarities and differences

- Both of these calculations are based on taking simple differences between neighboring points and adding them up over many steps, they're practically the same way to find a large sum of small differences
- We'll use arrays that contain data instead of calculating in terms of coordinates
- We'll need to find the neighbor differences over and over and over again, because each estimate only contributes to a small step forward

A tale of two substances

- Our example is from the study of *morphogenesis* (literally, “the creation of shapes” – it’s biology)
- Some cells produce two substances (activator and inhibitor) that support and resist a reaction that can change the cell when there’s enough of the activator
  - Change its color or something, say
- In time, these substances spill over to neighboring cells until they reach an equilibrium where they keep each other in check
- *Biological pattern formation: from basic mechanisms to complex structures* [A.J. Koch & H. Meinhardt, Reviews of Modern Physics vol. 66, No. 4] contains some equations that estimate how this happens

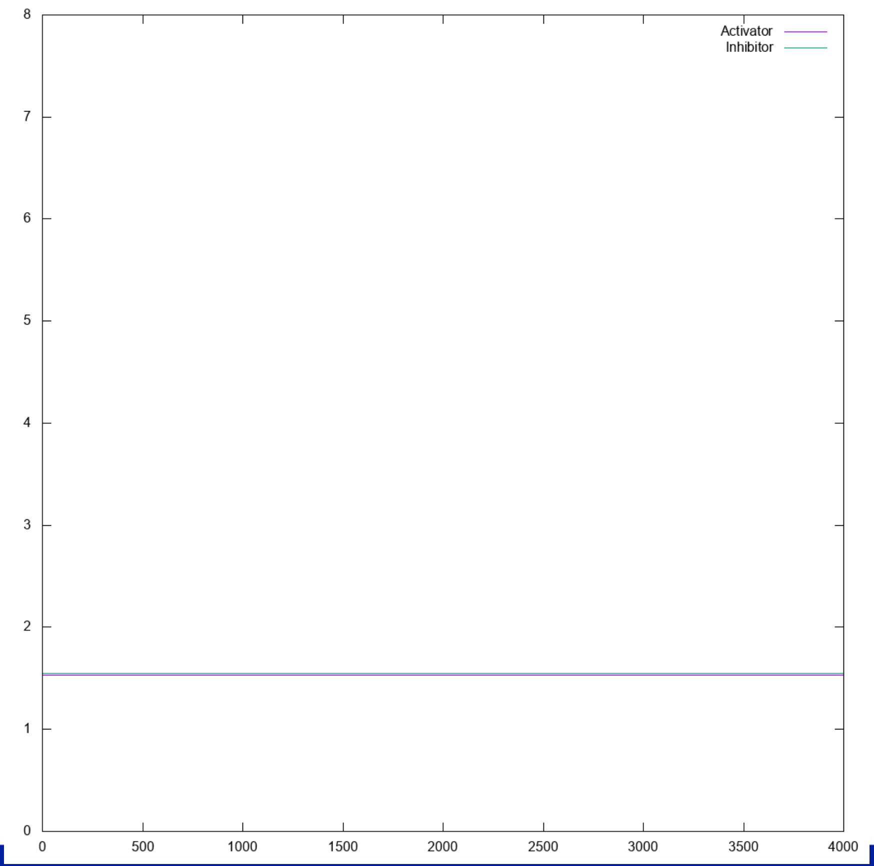
The integration scheme

- The equations are on lines 50-56 in the sequential example program *meinhardt_seq.c*
- We won’t derive them, I just took them from the paper
- As you can see, the time update is just a combination of elements at neighbor indices (n-1, n, n+1)

```
    // Approximate da/dt and db/dt with finite differences, integrate
    for ( int_t n=1; n<N-1; n++ )
    {
        real_t da_dt = DA * ( A(n-1) - 2.0*A(n) + A(n+1) );
        real_t db_dt = DB * ( B(n-1) - 2.0*B(n) + B(n+1) );
        A_nxt(n) = A(n) + (S*A(n)*A(n)+BA) / B(n) - RA * A(n) + da_dt;
        B_nxt(n) = B(n) + S*A(n)*A(n) - RB*B(n) + db_dt;
    }
```

Stalemate

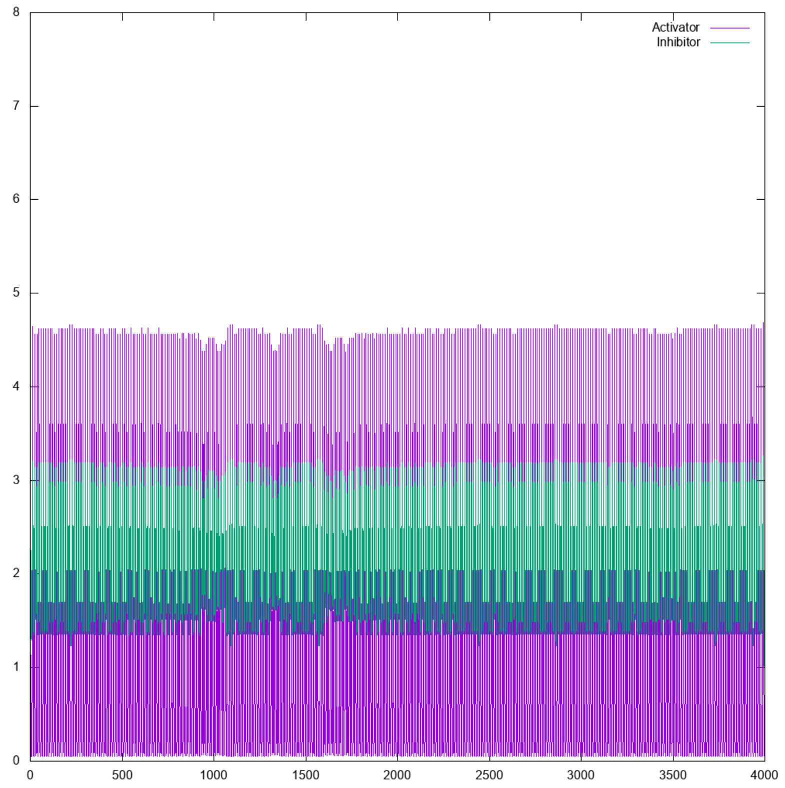
- When there's an equal amount of activator and inhibitor, the equilibrium state is pretty boring:



NTNU – Trondheim
Norwegian University of Science and Technology

Disorder

- If we disturb the equilibrium just at one little point, the whole system wobbles around for a while, before it settles in a very different stable state



NTNU – Trondheim
Norwegian University of Science and Technology

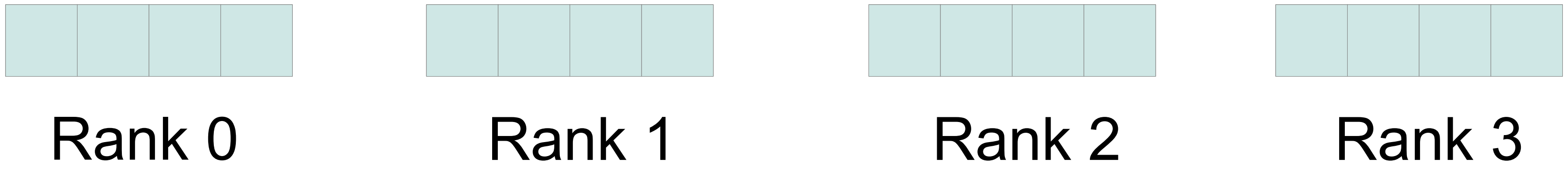
The Science^{TM}

- Different, tiny perturbations at the beginning lead to unpredictable final states
- They all look the same in a way, but they're different when you look in detail
- This is a small beginning of the explanation why all zebras have different but similar-looking stripe patterns, each was born with a different slightly random substance distribution in its skin

(We're honestly missing quite a lot of detail to get proper patterns here, but we ***are*** *simulating the main mechanism that drives the process...)*

Parallelizing with MPI

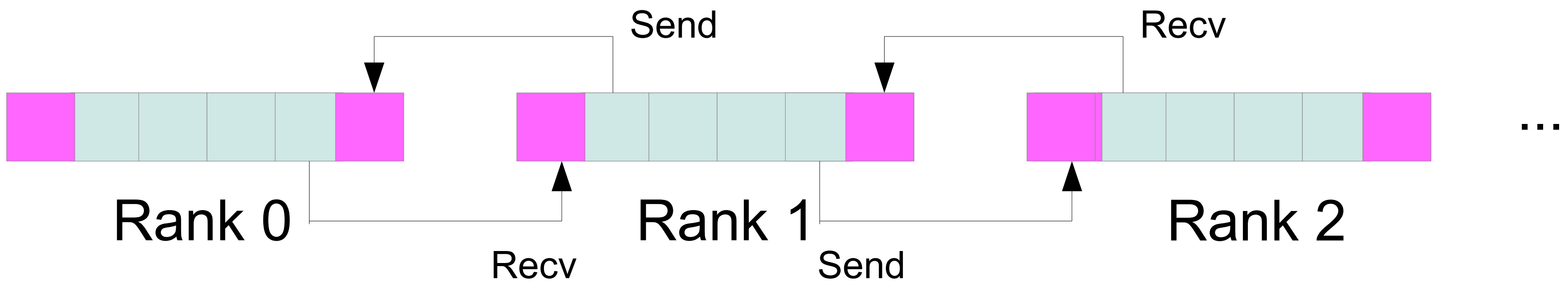
- More cells, more time steps, more dimensions all increase the time to equilibrium
- We can split the arrays and make separate cores work on their own, smaller parts for a faster result:



- This is *domain decomposition*, lots of number crunching programs contain variations of the theme

What about the borders?

- There's trouble where we divided the array: first and last cells need neighbor values from another process
- Solution: add *halo points* (aka. *ghost points*) to the arrays, and exchange their values for every step



After the exchange, each rank can use the extra elements as if they were locally computed

NTNU – Trondheim
Norwegian University of Science and Technology

Points from the code

- meinhardt_mpi.c splits the activator and inhibitor arrays from meinhardt_seq.c in this way, adding 1 point at each end
  - Indexing macros A,B are used so that indices can go from 0 to local_size-1, but still be valid for indices **-1** and **local_size**, outside the rank's own sub-array
  - Border exchange is in 4 MPI_Sendrecv calls, two for the activator, two for the inhibitor
- There's a use of MPI_Gather as well: rank 0 collects the whole system every now and again, and writes it into a file

Speedup

- With enough array points to compensate for the (somewhat slower) communication parts, this program
  - Allows faster solutions
  - Allows bigger problem sizes
- The file writing part is a bit stupid
  - Rank 0 is a bottleneck when it needs copies of everything, if it collects too often, the parallel version gets *slower* than the sequential
  - A much better solution is to use MPI_File_write which doesn't require a single collecting-rank to do all the work, but we don't have time to cover I/O facilities today

Summary

- We've spoken about
  - How MPI works under the hood, superficially
  - Building and running programs
  - Point-to-point messaging, deadlocks and synchronization
  - A handful of collective operations
  - Repeated integration over an x-axis
  - Border exchanges
- This mostly covers chapters 3-3.4 in the book

  ...and a slightly more complicated example.
  - Sorry about, but I *really really* wanted to demonstrate a problem with border exchanges in
  - This technique is important because it is practically everywhere in scientific computing, domain decomposition problems are 70-90% of the workload