

# Introduction To MPI

"The Angry Penguin", used under creative commons licence from Swantje Hess and Jannis Pohlmann.



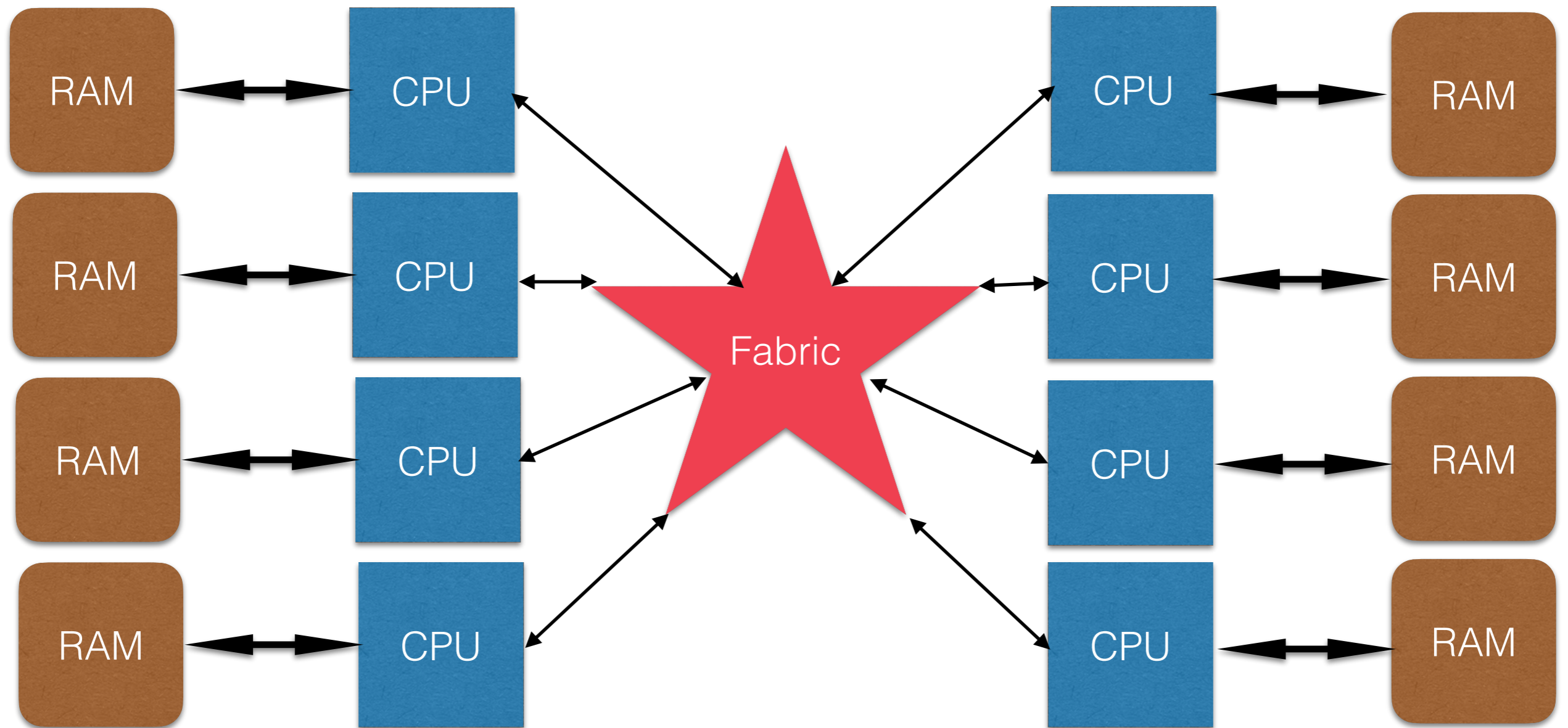
# Why MPI?

- The idea of MPI is to allow programs to communicate with each other to exchange data
  - Usually multiple copies of the same program running on different data - SPMD (single program multiple data)
- Usually used to break up a single problem to run across multiple computers
- Can be used for almost any parallel computing task, but most commonly used for tightly coupled parallelism across multiple computers
  - **Distributed memory parallelism**
- Actual MPI implementations are very efficient on a single machine as well but most people targeting a single machine don't use MPI

# Distributed Memory

- Processors all have their own memory
- Data can only flow between processors through a ***fabric***
- Typically send and receive data explicitly
  - Manual communication
- Synchronisation is tied directly to communication
  - When receive operation completes data is synchronised

# Distributed Memory



# Distributed Memory

- Have to manually work out the transfer of data between processors
- Send explicit **messages** between processors containing data
- This can be difficult in general but there are strategies
- Fabric is in general quite slow compared with memory access
  - Minimise the amount of data transferred and the number of transfers requested

# MPI

- The Message Passing Interface (MPI) is the most popular distributed memory library
- Just a library, no compiler involvement
- Includes routines for
  - Sending
  - Receiving
  - Collective operations (summing over processors etc.)
  - Parallel file I/O
  - Others

# InterProcess Comms

- MPI is not the only way of communicating between processors
  - "Pipes", "Unix Sockets"
  - However, like OMP it has been designed to support scientific types of program
    - Takes care of a lot of the details
    - You focus on what, when and where you communicate (it handles how)

# Parts of an MPI program

- Initialisation

- Compute

- Communication

- Output

- Finalisation

# Structure

- Note communication and compute are separate
- This is for several reasons
  - Clarity
    - It is difficult to debug and maintain code which mixes MPI code with general code
  - Performance
    - You want to do as little communication as possible in as few messages as possible

# Including MPI

- Since MPI is a library it has
  - an associated header file in C
  - an associated module in Fortran
  - import in Python
- `#include <mpi.h>`
- `USE mpi`
- `import mpi4py`
- Will not show actual code in these slides, but the additional material has examples in

# Initialisation & Finalization

- Before you can use MPI you have to call a routine called `MPI_Init`
- It is invalid to call any other MPI routines before `MPI_Init`
- After you have finished using MPI you must call `MPI_Finalize`
- Must call this before your code finishes or some MPI bits might not shut down properly

# Compiling and Running

- To test, have to compile (not Python) and run on more than one processor
- MPI library creates **compiler wrapper** that wraps your existing compiler
- Usually the compiler wrapper will be called mpicc (rather than gcc etc) or mpifort (previously mpif90) (rather than gfortran)
  - For commercial MPI libraries like Intel's, check the docs for the correct commands (mpiicc, mpiifort etc)
- To run on your local system use  
`mpiexec -n <number of processors> <program name>`
- On cluster resources, generally we use a scheduler aware program instead so we can just do `srun <program name>`

# MPI Features



# Important Ideas

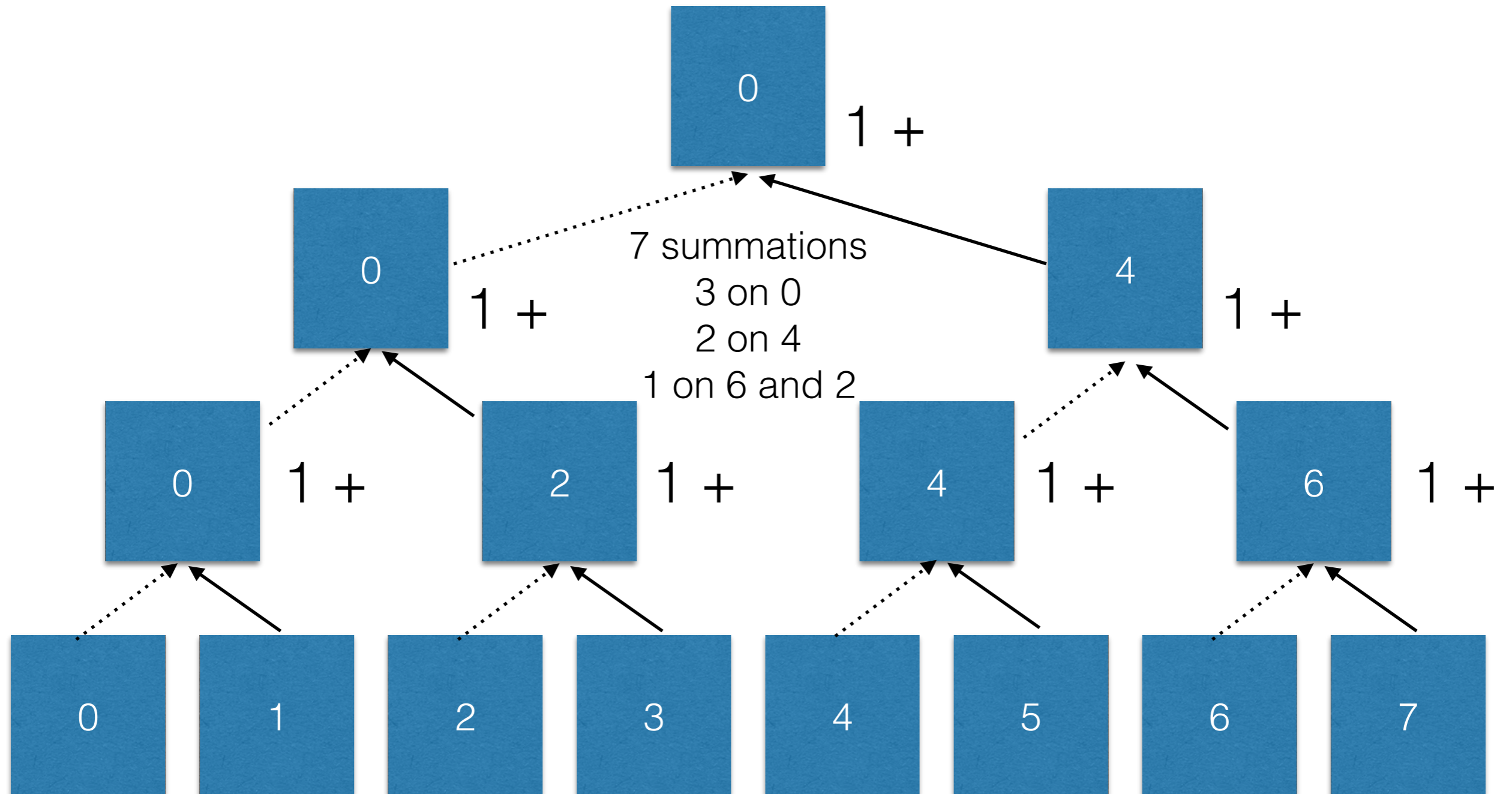
- One processor is root - usually 0
- All processors know that others exist
- You can (advanced) take subgroups of processors
- You have to explicitly communicate data by sending from a source and receiving on the destination
- You generally split data up across processors - it doesn't exist in any one place
  - There are **PGAS** models that allow you to act as though all of the data is in one place, but they are less common than MPI

# Collective Communication

- MPI has lots of routines for communication between ALL the processors involved
- Broadcast (send from one to all)
- Alltoall (send from all to all)
- Reduce (take data from all, combine it, send to one)
- Allreduce (take data from all, combine it, send to all)
- Scatter (root divides data and spreads to all)
- Gather (root receives data from all and concatenates)

# Processors in reduce

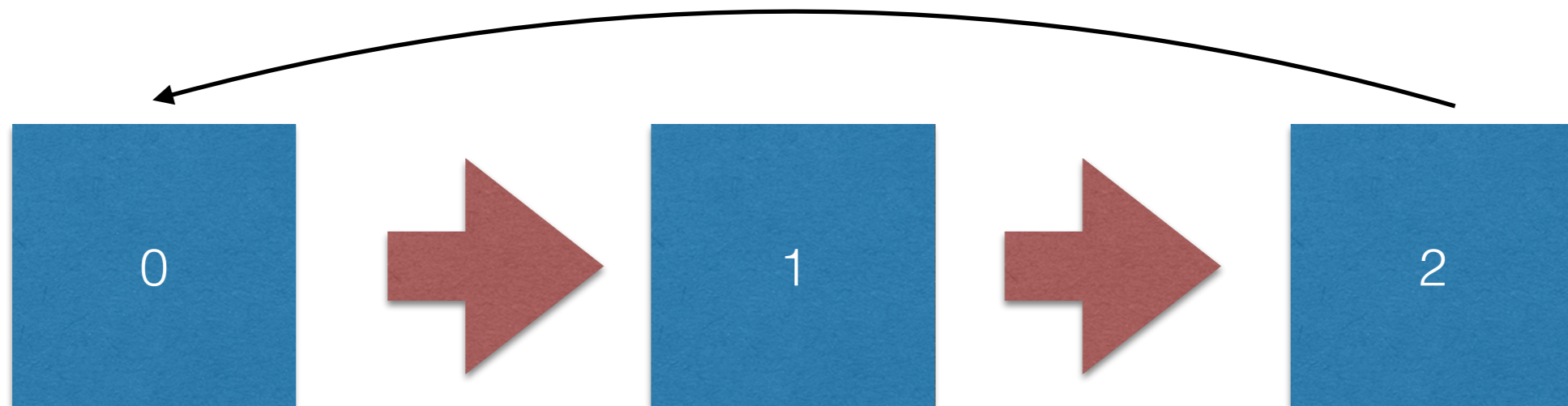
8 Processors = 7 messages to a range of processors



# Point to Point

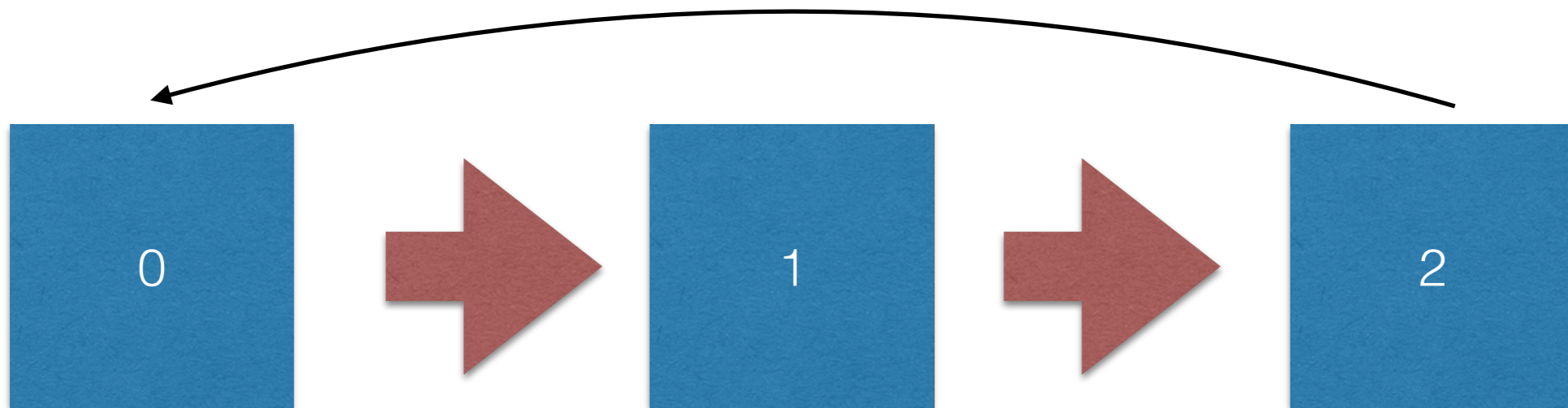
- MPI also supports point to point, or communication between one processor and one other
- Send, recv
- Imagine every processor has to pass one piece of data to the processor one higher, usually drawn as to the right
- At the top, it loops back to the 0th
- This is called the ring pass
- Send to the right, receive from the left

# Ring pass



- Rank 0 sends to 1
- Rank 1 sends to 2
- Rank 2 sends around to 0

# Deadlock



- Rank 0 calls MPI\_Send
- So does Rank 1 and Rank 2
- MPI\_Send BLOCKS - waits until done
- Nobody can call MPI\_Recv
- Nobody! - Deadlock

# Sendrecv

- Multiple solutions:
  - Beware serialising by accident
  - Red-black: odds send, evens recv
- MPI supports the whole process
  - MPI\_Sendrecv is a send and a receive bundled together and while both together block until they are finished, neither the send part nor the receive part blocks the other

# MPI Approaches



# Example problems

- Tons of example code in our MPI materials
- Here we'll just give some example of things that MPI does well
- Remember that we're probably using MPI because we need to go beyond a single computer or node

# Embarrassing Parallel

- We saw things like GNU parallel, Slurm job arrays etc which run many copies of a program with different inputs
- MPI is good at this too
- Each processor knows its in an MPI program and is given a number (its rank)
- Bonus - small amounts of shared work also possible

# Retrofitting the Unthreadable

- If you have non-thread-safe code, e.g. using a lot of global state
  - It might be easier (and less bug prone) to run as separate processes
  - Use MPI to handle the parts where it needs to communicate
  - Downside: separate copies of ALL of the memory
  - Downside: synchronisation is a cost; it may be hard to be efficient

# Worker Controller

- Worker Controller problems work well in MPI
- Write a single program
- Processor 0 (usual convention) will be controller
- Parcels up work, send messages to every other with their tasks
- Uses point-to-point comms almost entirely

# Domain Decomposition

- Imagine the image blurring on an image so big it won't fit on one processor
- Split it over many
- Blur on each separately
- Problem: processors share input data along the borders

# Domain Decomposition

- Red cells are the SAME cells on left and right
- Sharing this data is pretty much the ring-pass problem
- MPI\_sendrecv!

