Technologies

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

Warwick RSE

15/12/2023

Pthreads

Pthreads

- Stands for POSIX threads
  - All *nix operating systems support it (Linux, MacOS, AIX, other commercial Unixes)
- Not hard to use in theory but is very low level, slow to program and rather uncommon in academic programming

Simple Pthread example

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void* testthread(void* arg){

  int i =*(int*)arg;
  printf("Sleeping for %i\n", i);
  sleep(i);
  return NULL;

}

int main (int argc, char ** argv){

  int i;
  int ival[8];

  pthread_t mythreads[8];
  void* rvals[8];

  for (i = 0;i<8;++i){
    ival[i]=i;
    pthread_create(&mythreads[i], NULL, testthread, &ival[i]);
  }

  for (i = 0;i<8;++i){
    pthread_join(mythreads[i], rvals+i);
  }

}
```

Pthreads

- Pthreads is a very low level way of splitting work up over processors
- It is very time consuming to program complex programs using Pthreads
- If you just want to throw up a list of things to do and have them all run separately can be useful
    - Surprisingly annoying to work out how many processors you have

Other Threading Models

- A lot of more modern languages have their own threading models that lie on top of Pthreads
- They are all different but follow a similar pattern
    - Create threads
    - Recombine threads
    - Test thread states
    - Mutual Exclusion Objects

Note on Python

- Python is a very popular language but it is **not** well suited to parallel programming
- Python's native parallelism model is a threading model related to Pthreads
  - BUT the normal Python interpreter actually doesn't work in parallel! (Others like PyPy don't have this problem)
  - Your threads will all queue up one after the other until they leave Python code and enter an external library of some kind
  - Called the Global Interpreter Lock (GIL)
  - There is work going on to remove the GIL but it isn't clear when this is going to become mainstream

Note on Python

- If your code uses a lot of external library code
  - Or uses the Numba JIT compiler
- then threading might help you
- **Write as little Python code as you can!**
- If not then you might want to consider rewriting your code in Fortran or C/C++ first
  - You usually get at least 10x faster and often 100x or even 1000x

Note on Python

- There is no Python version of OpenMP
- There is a Python version of MPI
  - Called MPI4Py

OpenMP

OpenMP

- OpenMP is a combination of
  - a set of directives that tell the compiler how to parallelise bits of the code
    - Needs an OpenMP aware compiler to be respected but are ignored by a non OpenMP aware compiler
  - a library that gives your code access at runtime to information about the number of processors, how to split work up etc.
    - Will fail to compile on a non OpenMP aware compiler

OpenMP

- OpenMP allows for almost completely general parallel programming
- But by **far** the most common use of OpenMP is to split loops up so that different bits of the loop are handled by different processors
  - There's an obvious limitation to this: only loops that have independent iterations can be parallelised over
  - If you have a loop to advance a quantity in time then you **can't** parallelise that since iteration 2 depends on iteration 1, which depends on iteration 0 etc. etc.

OpenMP

```
PROGRAM loop_decompose

  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: nproc, i
  INTEGER, DIMENSION(:), ALLOCATABLE :: its_per_proc

  nproc = omp_get_max_threads()
  ALLOCATE(its_per_proc(nproc))
  its_per_proc = 0

!$OMP PARALLEL DO
  DO i = 1, max_its
    its_per_proc(omp_get_thread_num() + 1) = its_per_proc(omp_get_thread_num() + 1) + 1
  END DO
!$OMP END PARALLEL DO

  DO i = 1, nproc
    PRINT '(A, I0, A, I0, A)', 'Processor ', i, ' performed ', &
        its_per_proc(i), ' iterations'
  END DO

 PRINT '(A, I0)','Total work on all processors is ', SUM(its_per_proc)

END PROGRAM loop_decompose
```

OpenMP

- The directive OMP PARALLEL DO starts what is termed a **parallel region**
- OMP END PARALLEL DO ends the parallel region
- Inside parallel regions multiple processors will do work and you have to program in a suitable "parallel" way
- Outside parallel regions there is only one processor and everything is like normal programming

OpenMP C vs C++ vs Fortran

- OpenMP has many rather annoying features but one of the most annoying for teaching it is that it isn't quite the same in any of the supported languages. For example
- Fortran - OMP PARALLEL DO
- C/C++ - OMP PARALLEL FOR
- In Fortran you have both explicit Begin and End OpenMP markers, in C you use curly brackets like you do for native C block markers
- Our examples cover both but in the slides we're only covering Fortran because it is easier for C programmers to read Fortran than vice versa
- C++ is very like C but with a few differences

OpenMP

- The above example shows that all of your processors are working on the array
- And that all of the elements of the array are touched by one and only one processor
- The default number of threads is the number of **virtual** cores that your computer has
  - If your CPU has hyperthreading then it will be twice the number of actual cores. If not it will be the same as the number of actual cores

OpenMP

- The above example relies upon some default behaviour of OpenMP to give the right answer
- If you are splitting up the loop across processors what happens to the loop variable?
  - Each thread has a **private** copy of the loop variable
  - All other variables are **shared** between all threads
    - This isn't always what you want!

OpenMP

```
PROGRAM loop_decompose

  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: nproc, i, thread_id, its

!$OMP PARALLEL PRIVATE(its)
  its = 0
!$OMP DO
  DO i = 1, max_its
    its = its + 1
  END DO
!$OMP END DO

 PRINT '(A, I0, A, I0, A)', 'Processor ', omp_get_thread_num(), ' performed ', &
        its, ' iterations'
!$OMP END PARALLEL

END PROGRAM loop_decompose
```

OpenMP

- In this example I do two things
  - I split the OMP PARALLEL DO directive into two parts
    - OMP PARALLEL - Split into multiple threads
    - OMP DO - Automatically decompose a loop (as before)
  - I define that the variable **its** is PRIVATE so each thread has it's own copy of it

OpenMP

- OpenMP has four variable specifiers which you can apply to any OpenMP statement that starts a parallel region

  - PRIVATE - Each thread has it’s own version of the variable. **They do not have any particular value and do not inherit any value you gave the variable before starting the threads**

  - SHARED - The variable is the same on each thread. Changing a value on one thread will change it on the other threads

OpenMP

```
PROGRAM loop_decompose

  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: nproc, i, thread_id
  INTEGER, DIMENSION(:), ALLOCATABLE :: its_per_proc

  nproc = omp_get_max_threads()
  ALLOCATE(its_per_proc(nproc))
  its_per_proc = 0

!$OMP PARALLEL DO PRIVATE(thread_id)
  DO i = 1, max_its
    thread_id = omp_get_thread_num() + 1
    its_per_proc(thread_id) = its_per_proc(thread_id) + 1
  END DO
!$OMP END PARALLEL DO

  DO i = 1, nproc
    PRINT '(A, I0, A, I0, A)', 'Processor ', i, ' performed ', &
        its_per_proc(i), ' iterations'
  END DO

 PRINT '(A, I0)','Total work on all processors is ', SUM(its_per_proc)

END PROGRAM loop_decompose
```

OpenMP

- OpenMP has four variable specifiers which you can apply to any OpenMP statement that starts a parallel region
  - FIRSTPRIVATE - Like **PRIVATE** but any value that the variable had before entering the parallel region will be retained
  - LASTPRIVATE - Like **PRIVATE** but the last value that the variable had in the loop is retained after the parallel region

OpenMP

- The first example had every thread counting the number of iterations in a unique element of an array
  - The total number of iterations was then calculated by summing that array in serial
- The second example didn't even try to count the total number of iterations
  - Because simple ways of doing it won't work

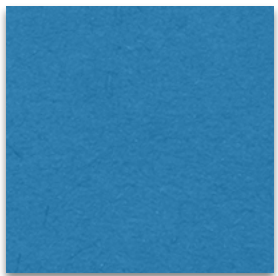
OpenMP

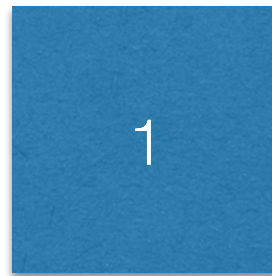
```
PROGRAM loop_decompose

  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: i, its_global


its_global = 0
!$OMP PARALLEL DO
  DO i = 1, max_its
    its_global = its_global + 1
  END DO
!$OMP END PARALLEL DO

PRINT '(A,I0,A)','Counter records ', its_global, ' iterations'

END PROGRAM loop_decompose
```

THIS DOESN'T WORK!!!!!

OpenMP

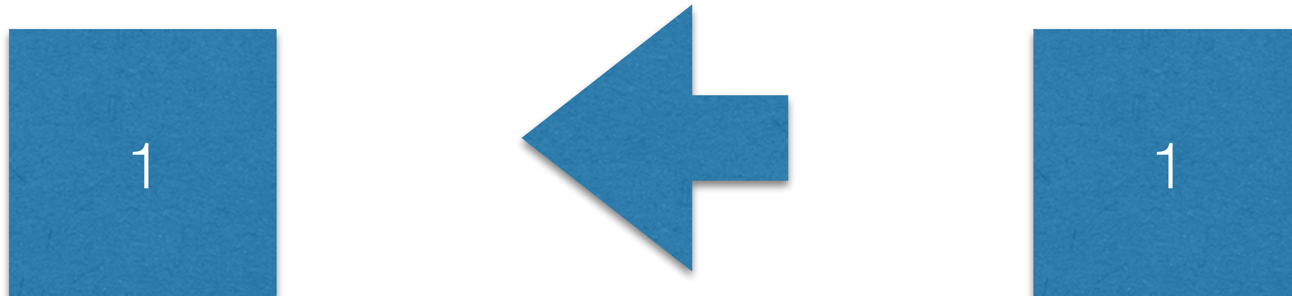
- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process





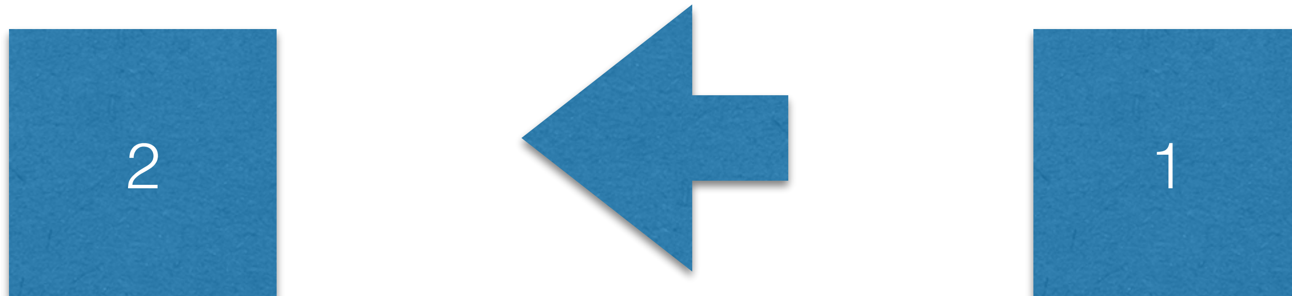
OpenMP

- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process



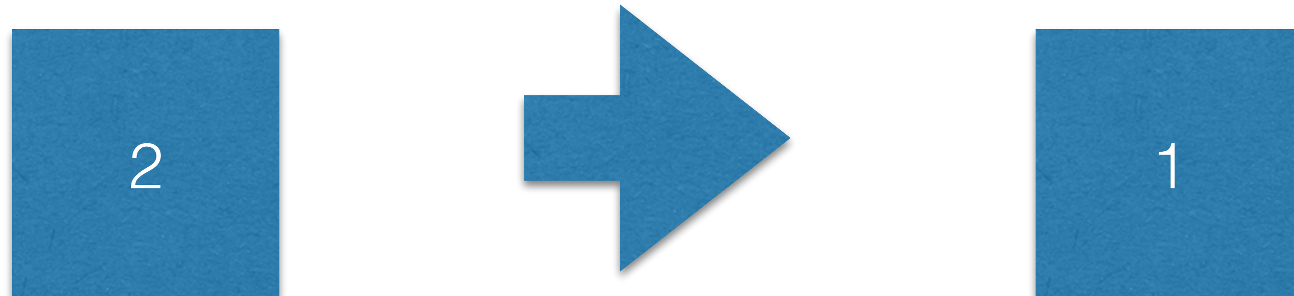
OpenMP

- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process



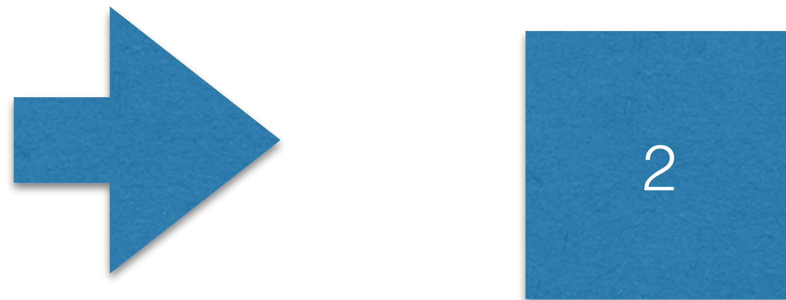
OpenMP

- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process



OpenMP

- **This is going to give the wrong answer!**
- This is because incrementing a counter isn’t an instantaneous process



OpenMP

- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process

OpenMP

- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process

OpenMP

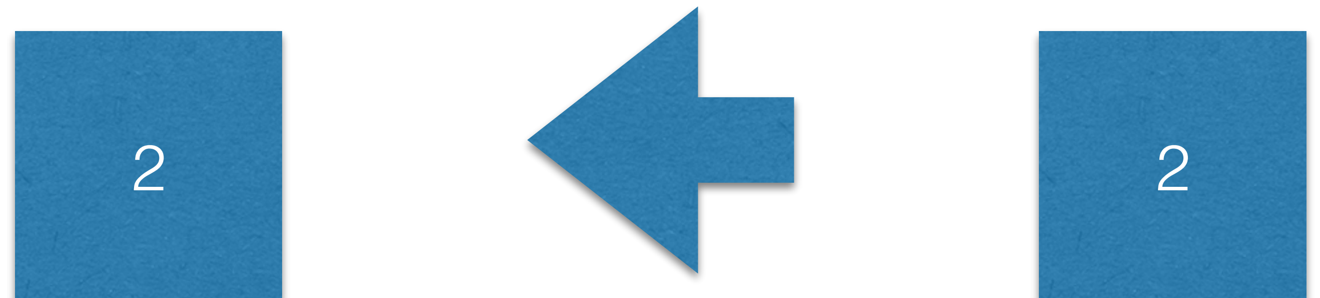
- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process

OpenMP

- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process

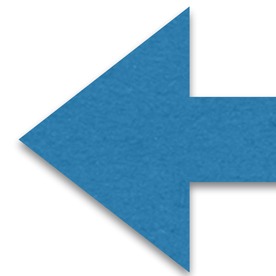
OpenMP

- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process



OpenMP

- **This is going to give the wrong answer!**
- This is because incrementing a counter isn't an instantaneous process



OpenMP

- There are three solutions to this. In order of increasing generality but decreasing efficiency they are
  - Atomic addition - Make that increment operation a single operation so that it can't be interrupted like that
  - Reduction - Tell OpenMP that you want to accumulate the local sum on each processor and then add them at the end
  - Critical sections - A region of the code that only one thread can be in at a time. c.f. Mutex

OpenMP

```
PROGRAM loop_decompose

  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: i, its_global

  its_global = 0
!$OMP PARALLEL DO
  DO i = 1, max_its
!$OMP ATOMIC
    its_global = its_global + 1
!$OMP END ATOMIC
  END DO
!$OMP END PARALLEL DO

PRINT '(A,I0,A)','Counter records ', its_global, ' iterations'

END PROGRAM loop_decompose
```

OpenMP

```fortran
PROGRAM loop_decompose

  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: i, its_global


its_global = 0
!$OMP PARALLEL DO REDUCTION(+:its_global)
  DO i = 1, max_its
    its_global = its_global + 1
  END DO
!$OMP END PARALLEL DO

PRINT '(A,I0,A)','Counter records ', its_global, ' iterations'

END PROGRAM loop_decompose
```

- Reduction variables are specified like SHARED or PRIVATE but have to include an reduction operator.
- Effectively it makes the variable PRIVATE but the reduction operator is applied at the end

OpenMP

```
PROGRAM loop_decompose

  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: i, its_global


its_global = 0
!$OMP PARALLEL DO
  DO i = 1, max_its
!$OMP CRITICAL
    its_global = its_global + 1
!$OMP END CRITICAL
  END DO
!$OMP END PARALLEL DO

PRINT '(A,I0,A)','Counter records ', its_global, ' iterations'

END PROGRAM loop_decompose
```

- Critical sections only ever have one thread in at a time
- The others have to wait their turn!

Other OpenMP

- There are a few other important OpenMP sections. All of them have to be in a parallel section to work
  - SINGLE - one and only one thread will go through this section
  - MASTER - thread 0 and only thread 0 will go through this section
  - WORKSHARE - Split up non-loop operations. Almost exclusively used for Fortran array operations

MPI

MPI

- **MPI is a library!**
  - You have to write code to use it
  - You have to have an installation of an MPI library to compile or run a program containing MPI code.
- You can write programs that compile with and without MPI support but you have to do something to remove or replace the MPI commands
- MPI is the most common way of programming for distributed cluster systems

MPI

- MPI is a set of standards, set out by the MPI forum (http://mpi-forum.org)
- Lots of library implementations of MPI but so long as you stay within the standard your code will work with **any** of them
    - Notwithstanding the fact that many strange and interesting bugs have cropped up in the libraries

MPI

- In general MPI programs are compiled using a **compiler wrapper** that is generated by the MPI library
  - Works exactly like the normal compiler from your perspective
- Normally (but by no means always) called
  - mpicc - C
  - mpic++ - C++
  - mpifort, previously mpif90 - Fortran

MPI

- You have to **Initialize** MPI before you can use it to communicate
- You have to **Finalise** it before your code finishes
- If you don’t do the first one then trying to use any other MPI commands will fail
- If you don’t do the second it is much more serious because it won’t (can’t!) catch it automatically and all kinds of errors can occur

First MPI code

```
USE mpi
PROGRAM main
  IMPLICIT NONE
  INTEGER :: errcode
  CALL MPI_Init(errcode)
  PRINT *, “Multiprocessor code”
  CALL MPI_Finalize(errcode)
END PROGRAM main
```

MPI

- MPI uniquely identifies processors using a number called the **rank**
  - Rank runs from 0 -> n_processors - 1
- You can subdivide processors in various ways using objects called **communicators**
  - There is a default communicator called **MPI_COMM_WORLD** that includes all processors
    - This is all that we will be using in this course but you can create others

Ranks and Processor Counts

```
PROGRAM main
  USE MPI
  IMPLICIT NONE
  INTEGER :: errcode
  INTEGER :: rank, nproc
  CALL MPI_Init(errcode)
  CALL MPI_Comm_size(MPI_COMM_WORLD, nproc, errcode)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, errcode)
  PRINT '(A, I3, A, I3)', 'I am processor ', &
      rank + 1, ' of ', nproc
  CALL MPI_Finalize(errcode)
END PROGRAM main
```

MPI

- MPI communications broadly break down into three chunks
  - Collective communications - processors all communicate in some way (usually to sum a quantity over all processors etc)
  - Point to Point communications - This processor talks to that processor
  - Weird stuff - very valuable but quite specific stuff

MPI

- Most MPI codes spend most of their time in point to point communication
- The rules aren't very complex
  - Sender sends a message to a specific recipient rank
  - Recipient receives a message, may be from a specific sender or from any sender
  - Messages have to be served in the order they arrive
  - The "normal" send and receive routines are **blocking**. They don't return until the message is sent or received (*)

Neighbour Pass

- The simplest possible MPI program is one that just sends a single data item to another processor
- We're going to do a simple thing where each processor sends its rank number to the processor with the next highest rank. The highest rank sends to rank 0

First Communications

```fortran
PROGRAM main

  USE MPI
  IMPLICIT NONE
  INTEGER :: rank, nproc, rank_right, rank_left, rank_recv, errcode
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: stat

  CALL MPI_INIT(errcode)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, errcode)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, errcode)

  rank_left = rank -1
  !Ranks run from 0 to nproc-1, so wrap the ends around to make a loop
  IF(rank_left == -1) rank_left = nproc-1
  rank_right = rank + 1
  IF(rank_right == nproc) rank_right = 0

  IF (rank == 0) THEN
    CALL MPI_Send(rank, 1, MPI_INTEGER, rank_right, 100, MPI_COMM_WORLD, &
        errcode)
    CALL MPI_Recv(rank_recv, 1, MPI_INTEGER, rank_left, 100, MPI_COMM_WORLD, &
        stat, errcode)
  ELSE
    CALL MPI_Recv(rank_recv, 1, MPI_INTEGER, rank_left, 100, MPI_COMM_WORLD, &
        stat, errcode)
    CALL MPI_Send(rank, 1, MPI_INTEGER, rank_right, 100, MPI_COMM_WORLD, &
        errcode)
  END IF

  PRINT ('("Rank ",I3," has received value ", I3, " from rank ", I3)'), &
      rank, rank_recv, rank_left

  CALL MPI_FINALIZE(errcode)
END PROGRAM main
```

Points to Note

- You specify
  - What variable to send/receive
  - How many elements of it there are (if it’s an array)
  - The type of the variable
  - Where to send to / receive from
  - An integer **tag** that has to match in send and receive (*) commands
  - The communicator that we are sending/receiving on

Points to Note

- Receiving has another parameter - the **status**
- This contains information about the message received
  - You're not usually interested in it though
  - There is a special value MPI_STATUS_IGNORE that you can use if you really don't care
- Fortran MPI takes an integer error code parameter at the end
- C MPI returns the error code as a return value

Deadlock

- It is tempting to just have every processor **Recv** it's message and then **Send** it on
  - This will in general **not work**
  - This is because **all** the processors will enter the receive and wait there because no-one is sending!
  - This is termed a **deadlock**

Deadlock

- Deadlock is caused (for example) when processor 1 is waiting for processor 0 to send and processor 0 is waiting for processor 1 to send
  - More complex versions are also possible
- Most of the runtime errors that you will encounter with MPI will be deadlocks
- There are also **livelocks** but they are much less common and rather harder to explain

Deadlock

- Be careful! Sometimes deadlocks only appear under certain conditions!
  - In particular **Send** commands actually return not when the message has been received but when the data being sent has been copied into the **send buffer**
  - If your message is small enough to fit into the buffer completely then **Send** will appear to return immediately
    - If your message gets big enough then this will suddenly stop happening and your program will deadlock
  - Sometimes running on a different system with a different configuration can trigger a deadlock
  - It is important to note that this is **not** an error on this system **it is an error in your code!**

Avoiding Deadlock

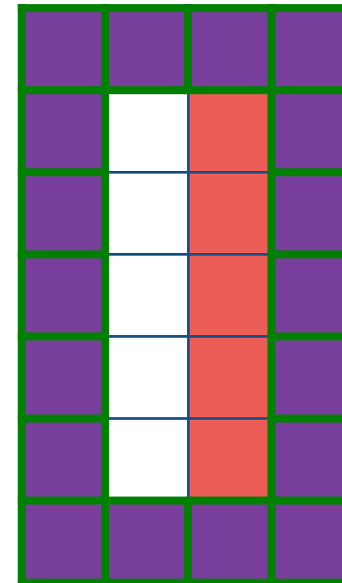
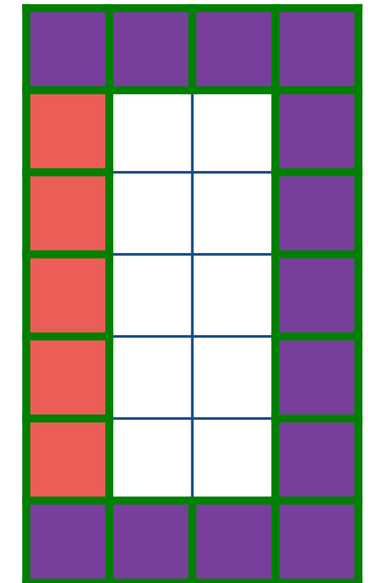
- There are various ways of avoiding deadlock
  - Non-blocking sends and receives (We're not covering these here)
  - MPI_SendRecv - combines a send and receive into a single command. The entire command blocks but the send and receive bits themselves can happen in any order
  - Plenty of others

First Communications

```
PROGRAM main

  USE MPI
  IMPLICIT NONE
  INTEGER :: rank, nproc, rank_right, rank_left, rank_recv, errcode
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: stat

  CALL MPI_INIT(errcode)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, errcode)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, errcode)

  rank_left = rank -1
  !Ranks run from 0 to nproc-1, so wrap the ends around to make a loop
  IF(rank_left == -1) rank_left = nproc-1
  rank_right = rank + 1
  IF(rank_right == nproc) rank_right = 0

  CALL MPI_Sendrecv(rank,     1, MPI_INTEGER, rank_right, 100, &
                    rank_recv,1, MPI_INTEGER, rank_left , 100, &
                    MPI_COMM_WORLD, stat, errcode)

  PRINT ('("Rank ",I3," has received value ", I3, " from rank ", I3)'), &
      rank, rank_recv, rank_left

  CALL MPI_FINALIZE(errcode)
END PROGRAM main
```

MPI_Sendrecv

- MPI_Sendrecv maps quite well onto that smoothing example that we showed earlier
- You send to your right and receive to your left
- When you reach an real edge of your problem you can use the special value **MPI_PROC_NULL** to not try to send or receive in that direction

Collective

- MPI collectives are not as big a part of most MPI codes as point to point communications but are still very important
- There are collectives to
  - combine data from processors (MPI_Reduce and MPI_Allreduce)
  - Send data from this processor to all other processors (MPI_Bcast, MPI_Scatter)
  - Get data from all other processors to this processor (MPI_Gather)
  - Send data from every processor to every other processor (MPI_Alltoall)
  - Synchronise all of the processors without sending data (MPI_Barrier)

Collective

```fortran
PROGRAM reduce

  USE MPI
  IMPLICIT NONE
  INTEGER :: nproc, rank, rank_red, errcode

  CALL MPI_Init(errcode)

  !Get the total number of processors
  CALL MPI_Comm_size(MPI_COMM_WORLD, nproc, errcode)

  !Get the rank of your current processor
  CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, errcode)

  !MPI_Reduce combines values from all processors. Here it finds the maximum
  !value (MPI_MAX) over all processors. It gets that value on one processor
  !called the "root" processor, here rank 0. The related MPI_Allreduce gives the
  !reduced value to all processors
  CALL MPI_Reduce(rank, rank_red, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD, &
      errcode)

  IF (rank == 0) PRINT *,'Largest rank is ', rank_red

  !MPI_Allreduce combines values from all processors. Here it finds the sum of
  !the value (MPI_SUM) over all processors. It gets that value on all processors
  CALL MPI_Allreduce(rank, rank_red, 1, MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD, &
      errcode)

  IF (rank == nproc-1) PRINT *, 'Sum of ranks is ', rank_red

  CALL MPI_Finalize(errcode)

END PROGRAM reduce
```

Collective

- All of the collectives are different to each other but they have commonalities
- They involve lots of processors interacting so they involve more communication than a single point to point message
- They are blocking (There are non-blocking variants but they are not used very commonly)