Programming with MPI

Introduction

Nick Maclaren

nmm1@cam.ac.uk

April 2010

Why Use MPI?

CPUs got faster at 40% per annum until ≈2003
Since then, they have got larger but not faster
The number of CPU cores per chip is now increasing

- The solution is to use more CPUs in parallel

MPI (Message Passing Interface) is a tool for that

We will come back to how to obtain MPI later

Before We Start

You must understand your language's basic features
What do you need? The course assumes very little

- Built–in datatypes and simple calculations
- The control statements and simple procedures
- Very simple 1– and 2–D arrays
- Very simple use of I/O and text files

A very few exercises need a little more

Course Structure (1)

Start with essential background and basic concepts
And running minimal but useful MPI programs

Then move on to facilities used in practice
Based on analysis of initially twelve real applications
Also mention features you might want in the future

Will describe their underlying concepts as relevant
Not well covered in most books and Web pages
This is helpful for debugging and tuning

Course Structure (2)

Also cover practical aspects that can cause trouble
Naturally, based on my personal experience!

Some of these (like I/O) are a bit weird
Will give simple guidelines for safe programming

Then give overview of more advanced features
Some are described in books and Web pages
But implementations may not be thoroughly tested

Will not go into detail for all of MPI

Applications

Applications I have looked at include:

> Casino, CASTEP, CETEP, CFX11, CPMD, CRYSTAL, DLPOLY_3, Fluent, FFTW, mpi_timer, ONETEP, PARPACK, SPOOLES ScaLAPACK and TOMCAT

Only facility course omits entirely is parallel I/O
Only in Fluent and DLPOLY_3 when I looked
Very specialist – few people will be interested

Course Objectives (1)

- The understanding of MPI's essential concepts
How it is likely to be implemented (in principle)

- Be able to use all basic features of MPI
For an empirical meaning of "all basic features"

- Be able to write highly parallel HPC code
Be able to work on almost all existing ones

- Be aware of the ancillary skills needed

Course Objectives (2)

- Be able to use I/O and other system interfaces
Including knowing something of what not to do

- Concepts needed for debugging and tuning
Some experience of doing so in simple programs

- Knowing what advanced features exist in MPI
So that you don’t have to reinvent the wheel

- Also knowing which features are tricky to use
So that you don’t use them by accident

Course Objectives (3)

- This teaches you to program MPI for real
It doesn't skip over anything you need to know
You will still have to look up some interfaces
The intent is that you know what to look up

- You will know why and how things work
Helps with writing reliable, portable code
Minimises confusion when you make a mistake
And gives a good start with tuning your code

All of the above is easier than it looks

Beyond the Course (1)

Contact your supervisor in the first instance
I am happy for your supervisor to contact me

The MPI standard home page – final authority
    http://www.mpi–forum.org/

Most books / courses skip over basic concepts
And too much time on the more advanced features

This one seems pretty good:
    http://www.cs.usfca.edu/mpi/

- This course does not follow it!

Beyond the Course (2)

The materials for this course are available from:

http://people.ds.cam.ac.uk/nmm1/MPI/

Several other relevant Computing Service courses
Some will be mentioned in passing, but see:

http://people.ds.cam.ac.uk/nmm1/

They are all ‘‘transferrable skills’’ courses
Not part of this MPhil, so get no credit

Beyond the Course (3)

All of these pages have reliable information
Most of the Web isn't reliable, of course

http://www-users.york.ac.uk/~mijp1/teaching/...
...4th_year_HPC/notes.shtml

http://www.epcc.ed.ac.uk/library/documentation/...
.../training/

http://www-unix.mcs.anl.gov/mpi/

Distributed Memory

One of the basic parallelism models

A program is run as separate, independent processes
Can be considered as separate serial programs

Distributed memory means no shared data

- The processes interact only by message passing

May be run on the same system or on separate ones

Message Passing

One of the basic communication designs

Process A sends a message to Process B
Process B then receives that message

- Think of it as process–to–process I/O or Email

Actually implemented using very similar mechanisms!

Some extra complications, but they use the same idea

What Is MPI? (1)

- A library callable from Fortran, C (and C++)
Bindings also available for Python, Java etc.

Primarily for HPC programs on multi–CPU systems
Assumes a number of processes running in parallel
Usually with dedicated CPUs (i.e. gang scheduling)

- Essentially all HPC work on clusters uses MPI
It works nearly as well on multi–core SMP systems

- Poorly for background work (e.g. cycle stealing)

What Is MPI? (2)

- It is a specialist communications library
Like POSIX I/O, TCP/IP etc. – but different purpose
Almost completely system–independent

- Using its interface is almost never a problem
If you can use any library, you can use MPI

- Most important step is to understand its model
I.e. the assumptions underlying its design
Ditto for C++, POSIX, Fortran, TCP/IP and .NET

The MPI Standard (1)

This was a genuinely open standardisation process
Mainly during the second half of the 1990s

http://www.mpi–forum.org/docs/docs.html

MPI 1 is basic facilities – all most people use
Most people use only a small fraction of it!

MPI 2 added extensions (other facilities)
Also included the MPI 1.3 update

MPI 3 adds yet more – not mentioned in this course

The MPI Standard (2)

- This is a standard, not a user's guide

Designed to be unambiguous, not easy to follow

As good as Fortran, much better than C or POSIX

- But its order and indexing are ghastly

⇒ I am still finding new features after a decade

- Use it to look up the precise specifications
- Use something else to find what to look up

Available Implementations

Two open source versions – MPICH and OpenMPI
You can install as packages or build from source
Most vendors have own, inc. Intel and Microsoft

Usually use shared–memory on multi–core machines
And TCP/IP over Ethernet and other networks
And often InfiniBand on suitable HPC clusters

- But NO code changes are needed!

MPI programs are very portable, and efficiently so

The MPI Model (1)

You start up N independent processes
All of them start MPI and use it to communicate

- There is no ‘‘master’’ (initial or main process)

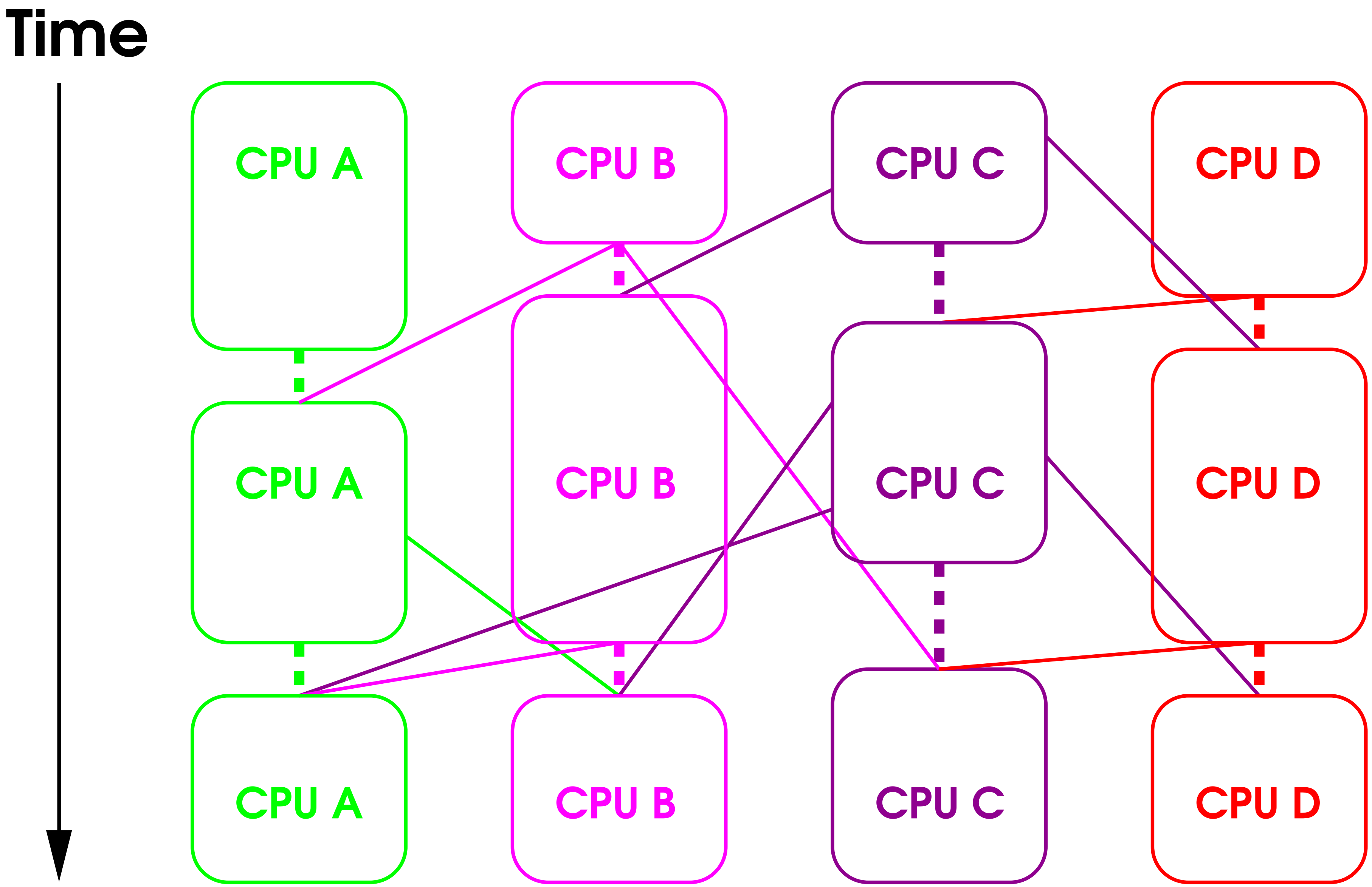
Communications may be ‘‘point–to–point’’ (pairwise)

- Only two communicating processes are involved

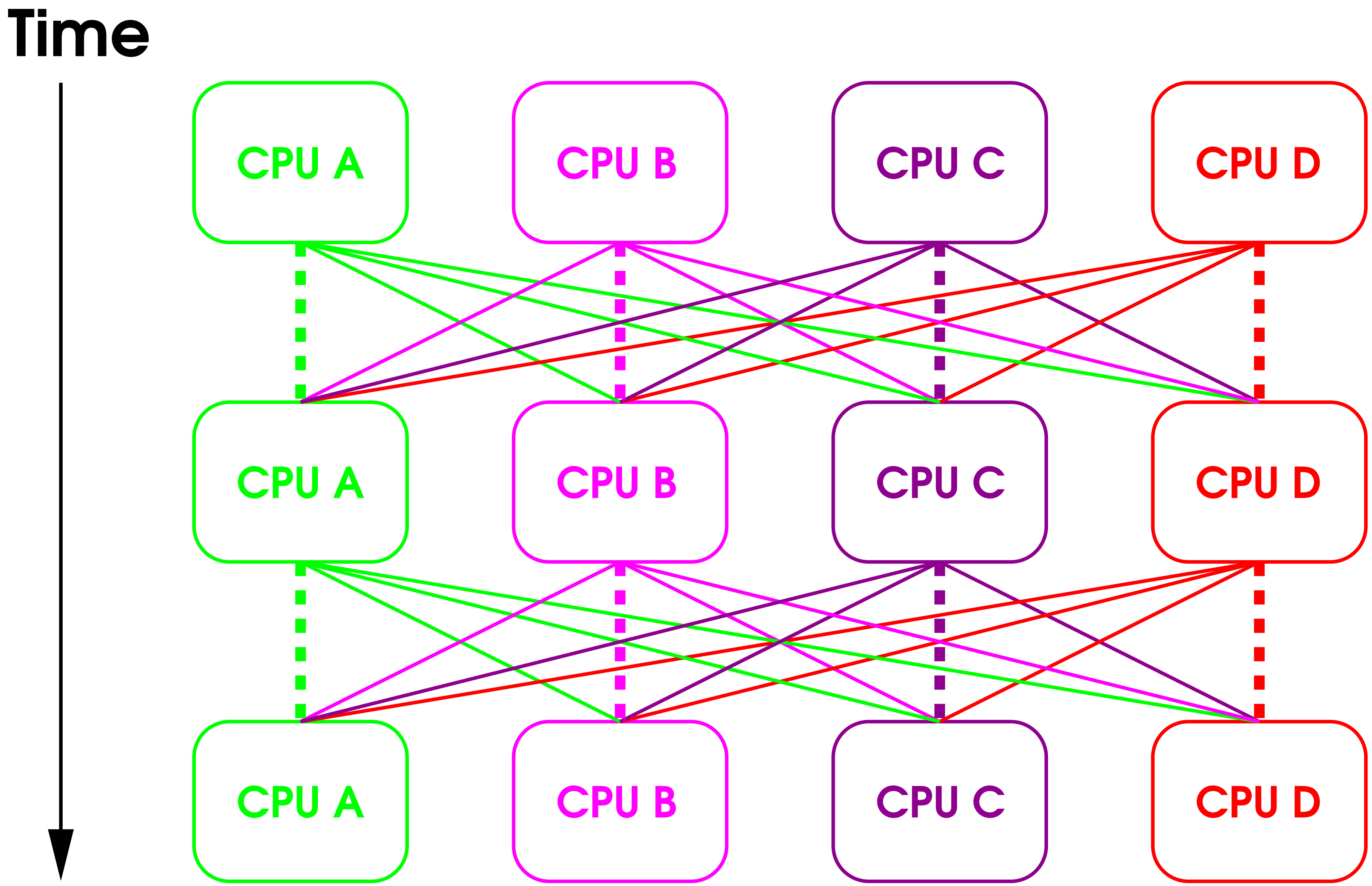
Communications may be ‘‘collective’’
All of the processes are involved

- They must all make the same call, together

Point to point Communication



Collective Communication



The MPI Model (2)

- Communication may not always synchronise
That applies to collectives as well as point–to–point
[The previous picture is misleading in that respect]

- Processes need wait only when they need data
E.g. a send may return before the receive
In theory, this allows for faster execution

- If you want synchronisation, you must ask for it
There are plenty of facilities for doing so

The MPI Model (3)

Some MPI operations are non–local
May involve behind–the–scenes communication
Which means they can hang if you make an error

And some operations are purely local
They can never hang, and will return ‘‘immediately’’

Generally, this matters mainly to MPI implementors

- You only need to know that both forms exist

The MPI Model (4)

- Almost everyone uses MPI in SPMD mode
  That is Single Program, Multiple Data
  You run N copies of one executable

- The programs can execute different instructions
  They don't have to run in lockstep (SIMD mode)
  That is Single Instruction, Multiple Data
  But start off by designing them to do that

- All CPUs are dedicated to your MPI program
  That avoids certain problems I won't describe now

The MPI Model (5)

SPMD isn't required by MPI, which surprises people
In theory, don't even need compatible systems
Could use it on a random collection of workstations

- Don't go there – and not because of MPI

For more detail on the reasons, see:

Parallel Programming: Options and Design

- This course will assume SPMD mode

Many implementations support only SPMD mode

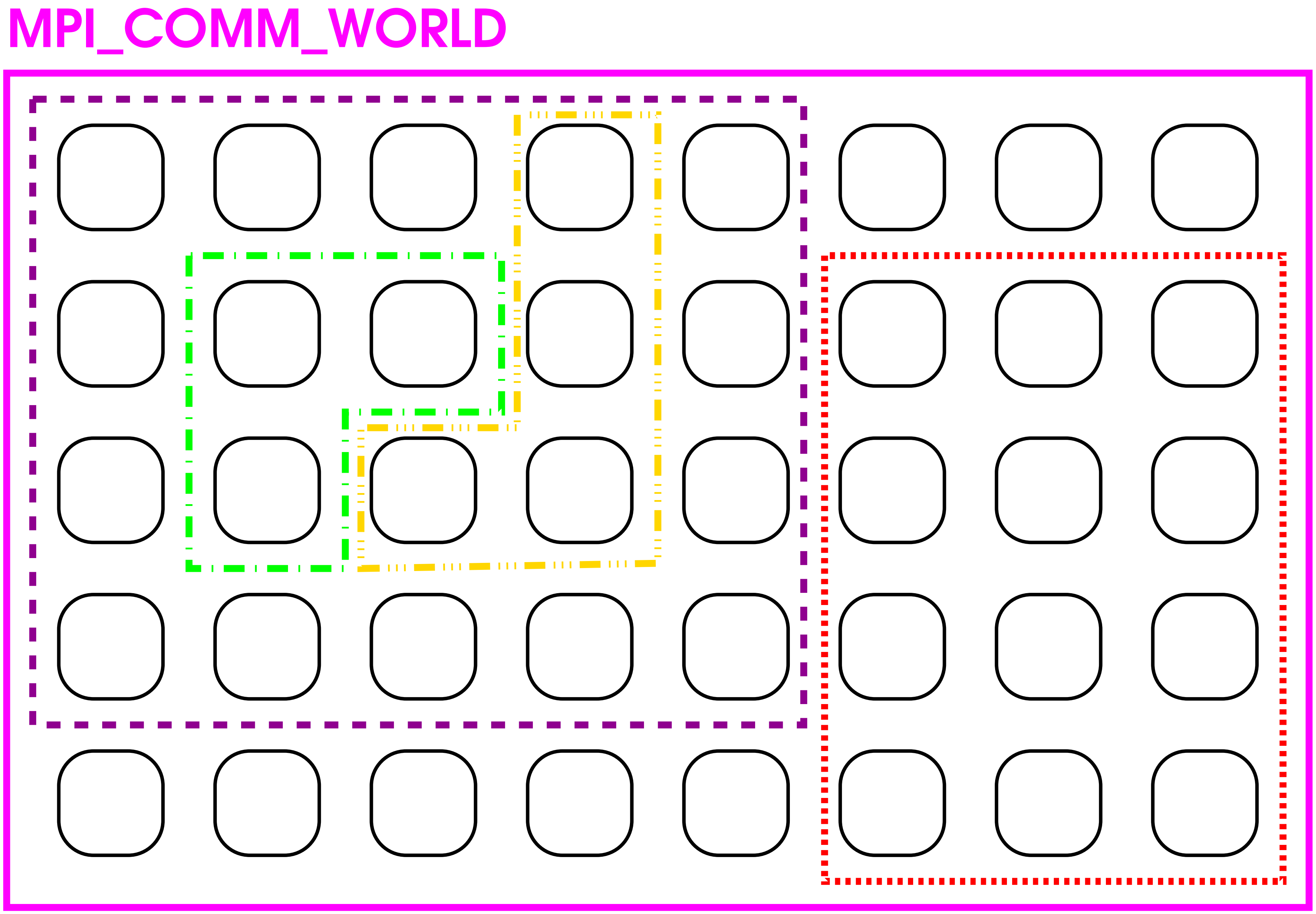
Communicators

- All communications occur within communicators
A context, defining a group of processes
Actions in separate communicators are independent

- You start with the communicator of all processes
You can subset any existing communicator

Facilities for that will be described later
- For now, use only MPI_COMM_WORLD

Hierarchical Communicators



Number of CPUs (1)

- Parallelism counting is “one, two, many”

You need to use different algorithms and code

One CPU is necessarily serial programming
Two CPUs are this CPU and the other CPU
Most issues arise only with many CPUs

- Serial codes may not work on many CPUs
- Parallel codes may not work on one CPU
- Two CPU codes may not work on either

Number of CPUs (2)

MPI communicators can have any number of CPUs
From zero CPUs upwards – yes, no CPUs

Use 4+ CPUs when debugging generic MPI codes
- Most applications assume at least that many

This course will cover only this case

Otherwise, you need different code for:

- 0: typically do nothing (not executed!)
- 1: use serial code for this
- 2–3: a few generic algorithms fail
- 4+: 'proper' parallel working

Diversion – a Worked Example

Shall now give a worked example of the use of MPI
Calculate the area of the Mandelbrot set

This is to give a feel for what MPI is about
Don’t worry if you don’t understand the details
Every facility used will be explained later

- The whole source is in the extra files

There are Fortran 90, C and C++ versions

The Mandelbrot Set

This is defined in the complex plane

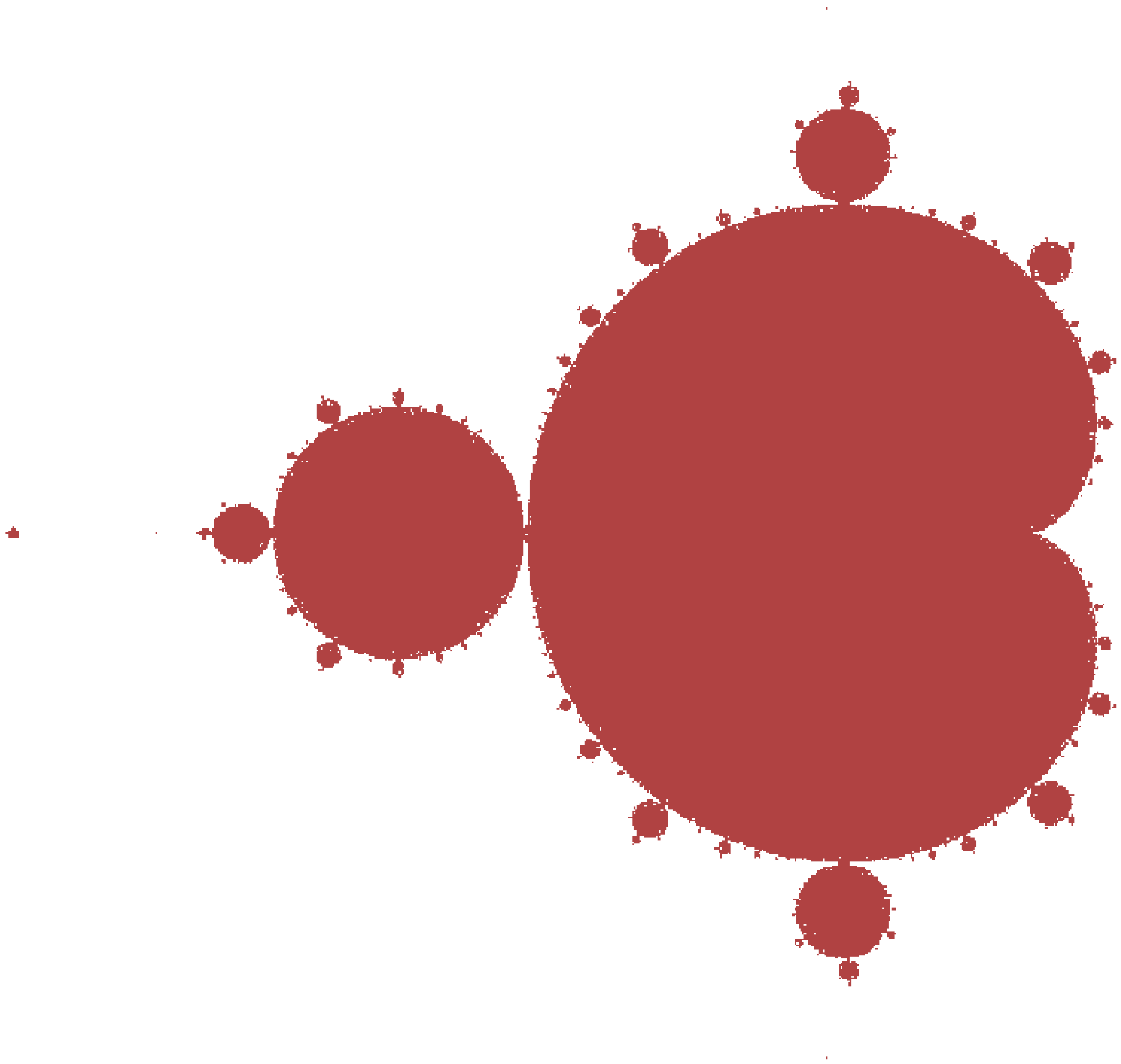
Consider the recurrence $x_{n+1} \Leftarrow x_n^2 + c$

With the starting condition $x_0 = 0$

The Mandelbrot set is the set of all c, such that

$|x_n| \leq 2$, for all n

This is, er, complicated – let's see a picture



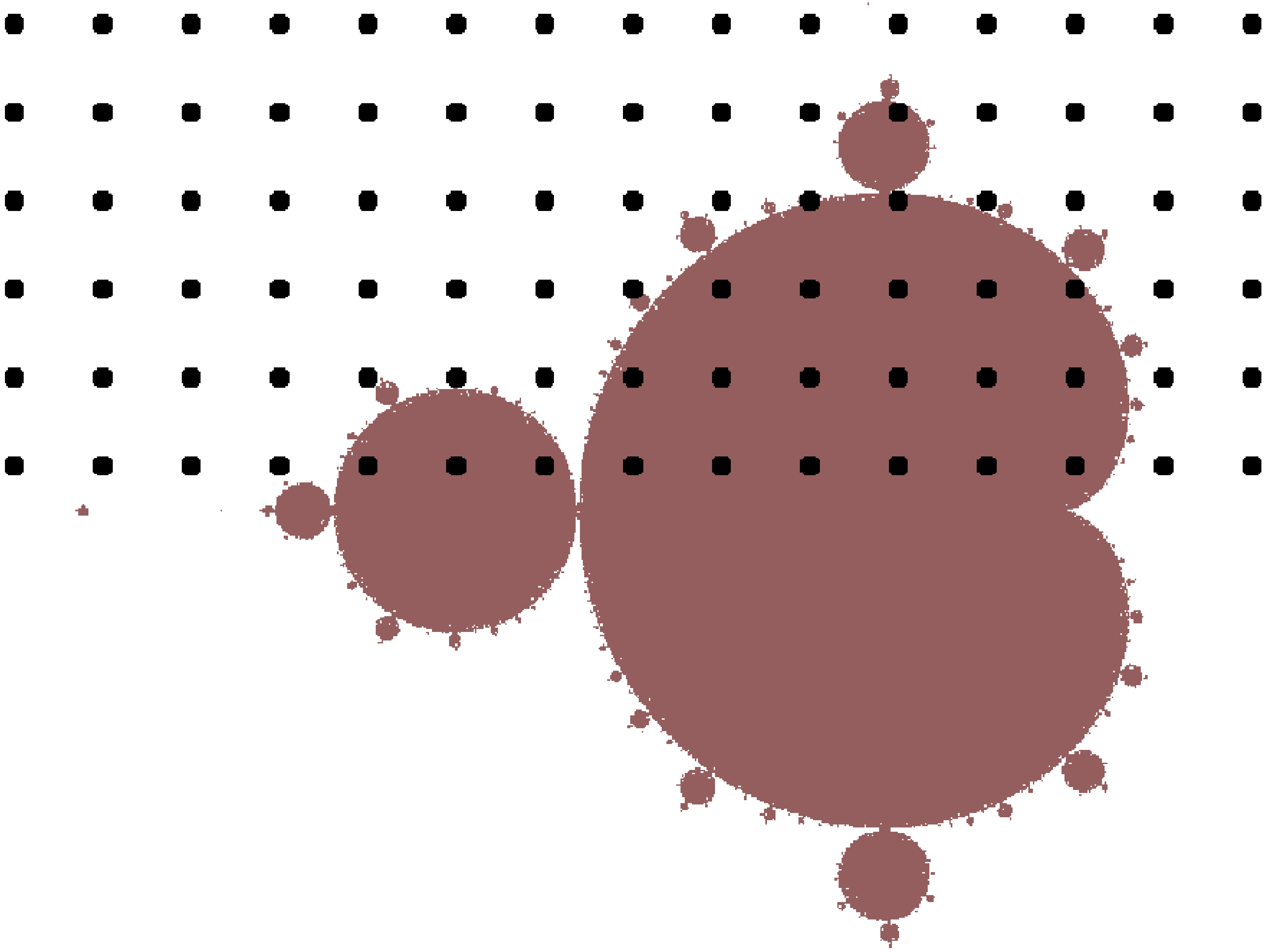
Calculating its Area

All points within it have $|c| \leq 2$
It's also symmetric about the X–axis

So we consider just points c, such that

$$-2 < re(c) \leq +2$$
$$0 < im(c) \leq +2$$

Choose a suitable iteration limit and step size
See if each point stays small for that long
Accumulate the scaled count of those that do



Example Program

This is the crudest form of numerical integration
Not strictly Monte–Carlo, but is related
Sometimes a sledgehammer is the best tool!

I have chosen to use Fortran 90
The C or C++ are very similar

Most of it is just the ordinary, serial logic
I will go through the core of it first

Testing a Point

```
PURE FUNCTION Kernel (value)
    IMPLICIT NONE
    LOGICAL :: Kernel
    COMPLEX(KIND=DP), INTENT(IN) :: value
    COMPLEX(KIND=DP) :: work
    INTEGER :: n

    work = value
    DO n = 1, maxiters
        work = work**2 + value
        IF (ABS(REAL(work)) > 2.0 .OR.        &
            ABS(AIMAG(work)) > 2.0) EXIT
    END DO
    Kernel = (ABS(WORK) <= 2.0)
END FUNCTION Kernel
```

Scanning an Area

```
PURE FUNCTION Shell (lower, upper)
    IMPLICIT NONE
    REAL(KIND=DP) :: Shell
    COMPLEX(KIND=DP), INTENT(IN) :: lower, upper
    COMPLEX(KIND=DP) :: work

    Shell = 0.0_DP
    work = CMPLX(REAL(lower),      &
        AIMAG(lower)+step/2.0_DP,KIND=DP)
    DO WHILE (AIMAG(work) < AIMAG(upper))
        DO WHILE (REAL(work) < REAL(upper))
            IF (Kernel(work)) Shell = Shell+step**2
            work = work+step
        END DO
        work = CMPLX(REAL(lower),AIMAG(work)+step,KIND=DP)
    END DO
END FUNCTION Shell
```

MPI Initialisation

```
LOGICAL, PARAMETER :: UseMPI = .True.
INTEGER, PARAMETER :: root = 0
INTEGER :: maxiters, error, nprocs, myrank
REAL(KIND=DP) :: buffer_1(2), step, x

IF (UseMPI) THEN
      CALL MPI_Init(error)
     CALL MPI_Comm_size(MPI_COMM_WORLD,nprocs,error)
    CALL MPI_Comm_rank(MPI_COMM_WORLD,myrank,error)
ELSE
      nprocs = 1
      myrank = root
END IF
```

Divide Area into Domains

```
COMPLEX(KIND=DP), ALLOCATABLE :: buffer_2(:,:)

IF (myrank == root) THEN
    ALLOCATE(buffer_2(2,nprocs))
    buffer_2(1,1) = CMPLX(-2.0_DP,0.0_DP,KIND=DP)
    DO i = 1,nprocs-1
        x = i*2.0_DP/nprocs
        buffer_2(2,i) = CMPLX(2.0_DP,x,KIND=DP)
        buffer_2(1,i+1) = CMPLX(-2.0_DP,x,KIND=DP)
    END DO
    buffer_2(2,nprocs) = CMPLX(2.0_DP,2.0_DP,KIND=DP)
ELSE
    ALLOCATE(buffer_2(2,1))   ! This is not actually used
END IF
```

Reading the Parameters

```
INTEGER :: maxiters
REAL(KIND=DP) :: step

IF (myrank == root) THEN
    READ *, maxiters, step
    IF (maxiters < 10) THEN
        PRINT *, 'Invalid value of MAXITERS'
        CALL MPI_Abort(MPI_COMM_WORLD,1,error)
    END IF
    IF (step < 10.0_DP*EPSILON(step) .OR. step > 0.1_DP) THEN
        PRINT *, 'Invalid value of STEP'
        CALL MPI_Abort(MPI_COMM_WORLD,1,error)
    END IF
END IF
```

Distribute the Data (1)

```
REAL(KIND=DP) :: buffer_1(2)
COMPLEX(KIND=DP), ALLOCATABLE :: buffer_2(:,:)
COMPLEX(KIND=DP) :: buffer_3(2)

IF (myrank == root) THEN
        buffer_1(1) = maxiters
        buffer_1(2) = step
END IF
```

Distribute the Data (2)

```
IF (UseMPI) THEN
        CALL MPI_Bcast(        &
                buffer_1,2,MPI_DOUBLE_PRECISION,        &
                root,MPI_COMM_WORLD,error)
        maxiters = buffer_1(1)
        step = buffer_1(2)
        CALL MPI_Scatter(        &
                buffer_2,2,MPI_DOUBLE_COMPLEX,        &
                buffer_3,2,MPI_DOUBLE_COMPLEX,        &
                root,MPI_COMM_WORLD,error)
ELSE
        buffer_3 = buffer_2(:,1)
END IF
```

Accumulate in Parallel

```
buffer_1(1) = Shell(buffer_3(1),buffer_3(2))
IF (UseMPI) THEN
      CALL MPI_Reduce(       &
            buffer_1(1),buffer_1(2),       &
            1,MPI_DOUBLE_PRECISION,       &
            MPI_SUM,root,MPI_COMM_WORLD,error)
ELSE
      buffer_1(2) = buffer_1(1)
END IF
```

Print Results and Terminate

```
IF (myrank == root) THEN
    PRINT '(A,F6.3)',        &
        'Area of Mandelbrot set is about',        &
        2.0_DP*buffer_1(2)
END IF
IF (UseMPI) THEN
    CALL MPI_Finalize(error)
END IF
```

So What Happens?

Running with parameters '10000 0.001'
We get about 1.508 (true result is about1.506)

Number of processors	Elapsed time taken
1	67 seconds
4	46 seconds
16	23 seconds

Not very scalable, is it? That is quite common
Using MPI is much easier than tuning it

Doing Better (1)

There is an alternative Fortran 90 version, too
Generates all of the points and randomises them
Each processor has a roughly matching workload

It is a store hog, and takes some time to start

Number of processors	Elapsed time taken
1	70 seconds
4	19 seconds
16	8 seconds

It would scale better with more points

Doing Better (2)

There is a better way than even that, too
Covered in the Problem Decomposition lecture
The first practical of that gets you to do it

Suitable for embarrassingly parallel problems
E.g. parameter searching and Monte–Carlo work
Mandelbrot set was merely a convenient example

But that’s a lot later . . .