

# Programming with MPI

*Communicators etc.*

Nick Maclaren

[nmm1@cam.ac.uk](mailto:nmm1@cam.ac.uk)

May 2008

# Basic Concepts

A **group** is a **set** of process **identifiers**

Programs view them as integers **0...(size-1)**

A **context** is the **communication environment**

Separate **contexts** are entirely independent

Programs don't (and can't) view **contexts** directly

A **communicator** is a **group** plus a **context**

So separate **communicators** are independent, too

- Even if they have the same **group** of processes

Normally, we work solely on **communicators**

# Predefined Communicators

There are several predefined **communicators**  
Use these when appropriate

**MPI\_COMM\_WORLD** is all **processes** together

**MPI\_COMM\_SELF** is just the **local process**

**MPI\_COMM\_NULL** is an **invalid communicator**  
Used as an **error result** from several functions

# Use of Communicators (1)

Most people use only `MPI_COMM_WORLD`

We covered information calls in the first lecture

`MPI_Comm_rank` and `MPI_Comm_size`

Why do we need to go beyond that?

- To use `collectives` on only some `processes`
- Need to do a task on only some `processes`
- Want to do several tasks in parallel

Can do those messily by using `point-to-point`

Or by creating new, `subset communicators`

# Use of Communicators (2)

Avoid using two **communicators** that **overlap**  
Including one together with a **subset** of itself  
Clean up the use of one before starting the other

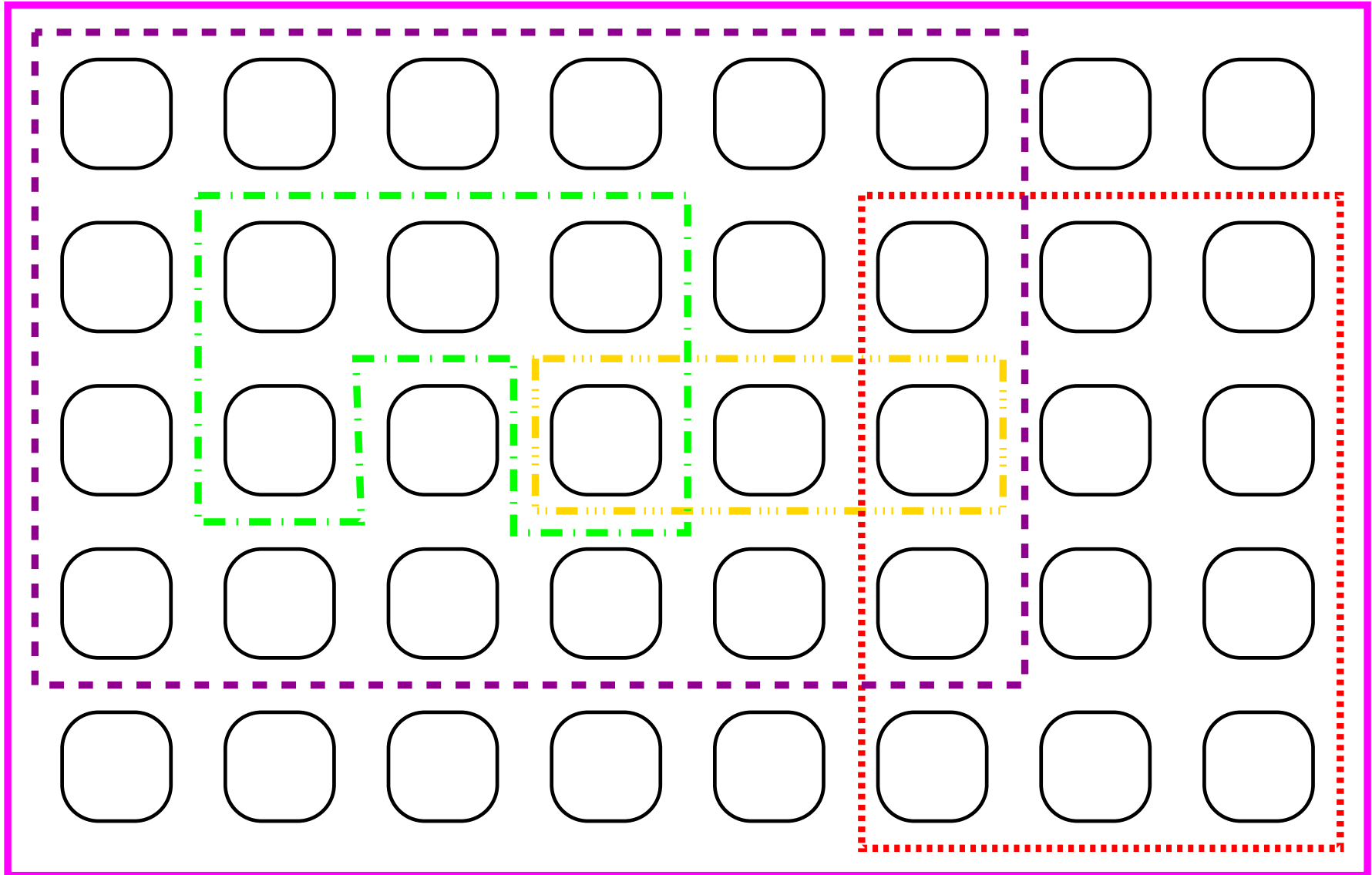
- MPI won't get confused – but you and I will  
And don't even think of trying to tune such a mess!

Design your **communicator** use to be **hierarchical**  
Like **recursion** in **groups** of **processes**

This is easier to show using pictures

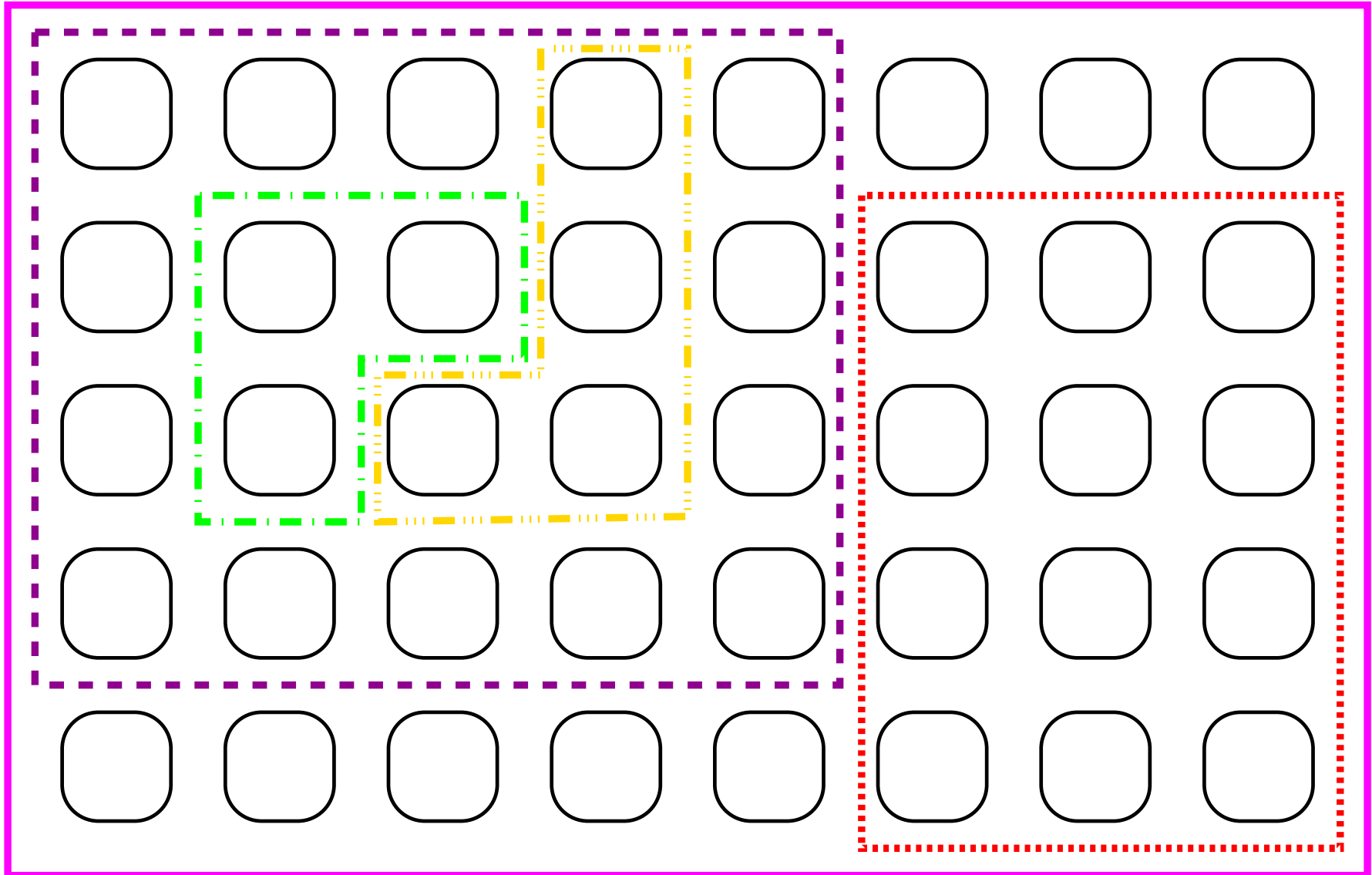
# General Communicators

**MPI\_COMM\_WORLD**

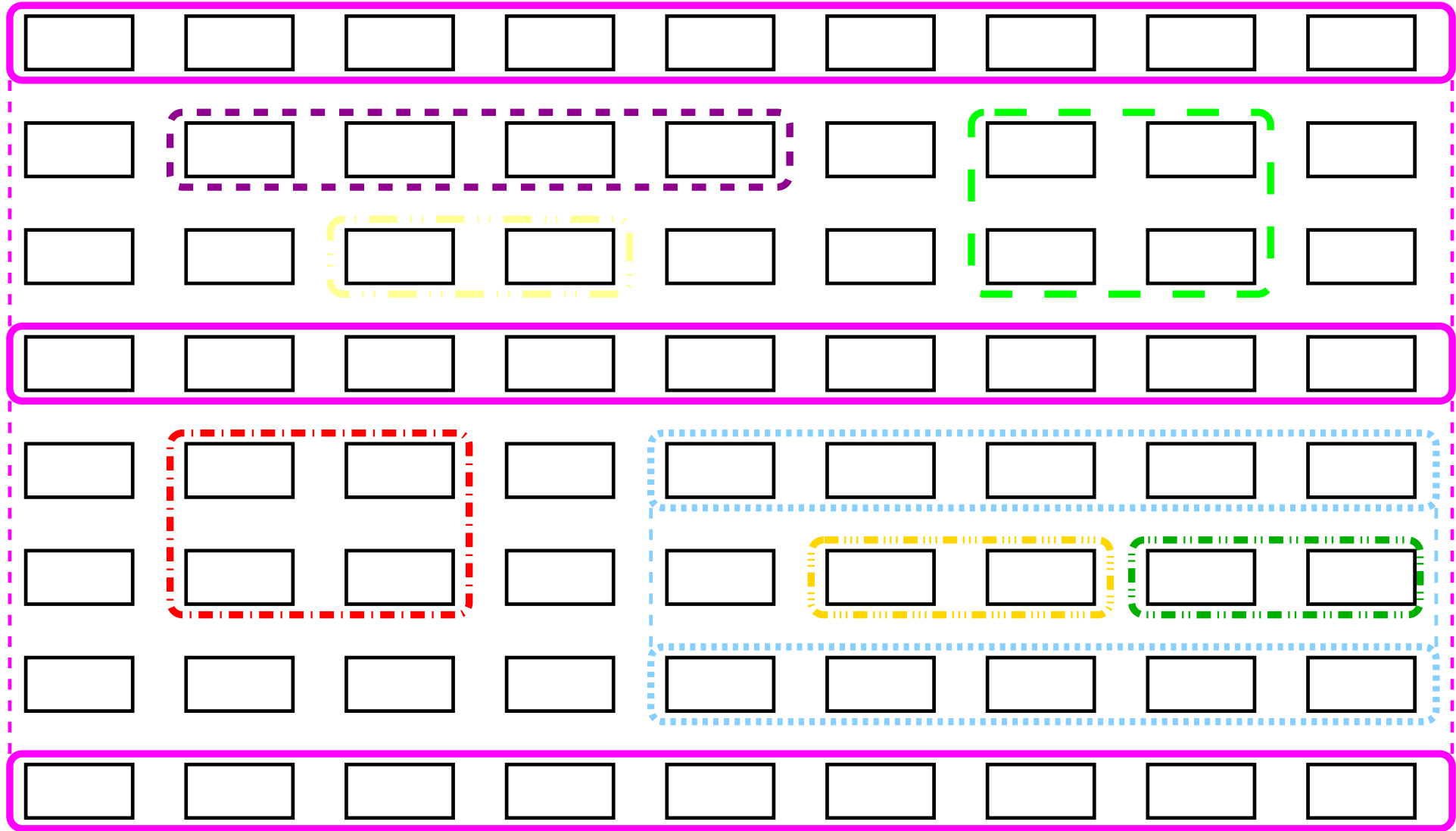


# Hierarchical Communicators

**MPI\_COMM\_WORLD**



# Using Hierarchies





# Splitting Communicators (1)

- You always start with an existing **communicator**  
And **subdivide** it to make one or more new ones  
A **collective** call on the existing **communicator**

- Each **process** specifies a **non-negative integer**  
The **value** is commonly called the **colour**  
Each new **communicator** corresponds to one **colour**  
E.g. all **processes** that specify the integer **42**

If two processes specify **different colours**  
the call returns **different communicators**

- A **communicator** is a **value** not an identifier

# Splitting Communicators (2)

Can also specify `MPI_UNDEFINED` to opt out

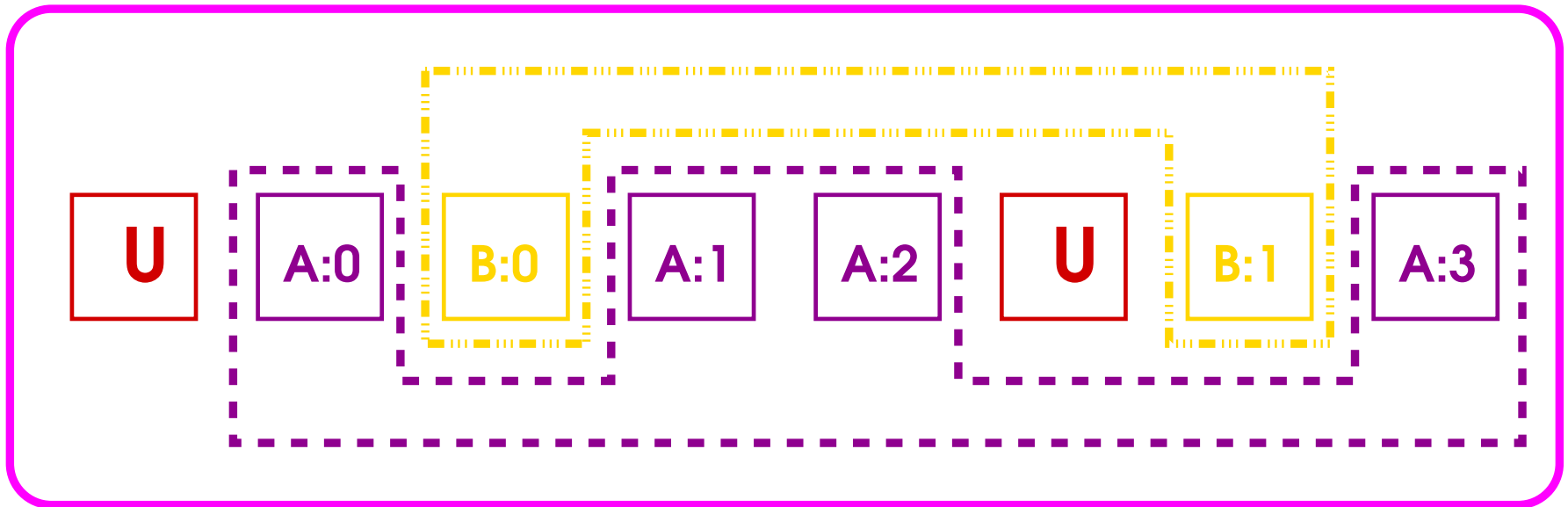
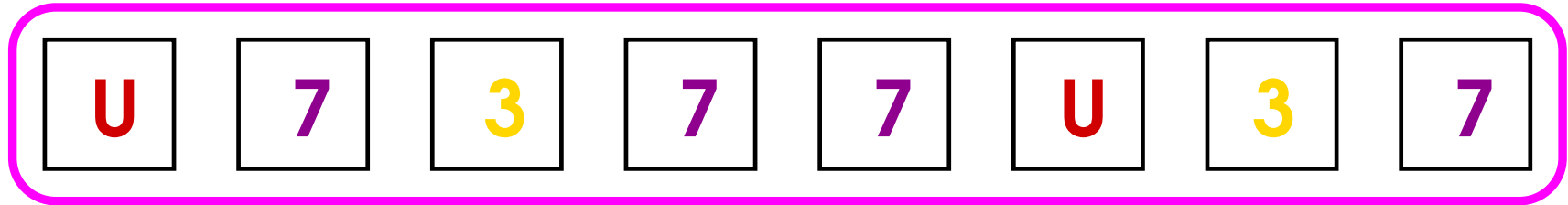
That is an unspecified `negative integer`

Note that `zero` is a `valid colour`

Call will return `MPI_COMM_NULL`

- This is an `invalid communicator` – don't use it

# Splitting Communicators



# Splitting Communicators (3)

Can also set the **rank** in the new **communicator**

A **key** argument that has an **integer value**

Any values are allowed, even **negative** ones

**Processes** have **ranks** in **key** order

All **keys** to **zero** says you don't care

- I recommend doing just that – one less detail

Doing anything else with **keys** is advanced use

Comparable to operating on **groups** directly

# Destroying Communicators

When you have finished with a **communicator**  
You should **free** (**delete/destroy**) it  
A **collective** call on the **communicator**

This will free any resources it uses

- You **must** tidy up all **transfers** first  
Some **libraries** and **tools** may check that is so
- You needn't free it if you only stop using it  
I.e. when you are going to **reuse** it later

# Split (1)

Fortran example:

```
INTEGER :: colour , newcomm , error
!   'colour' is set to an appropriate value

CALL MPI_Comm_split (    &
    MPI_COMM_WORLD ,    &
    colour , 0 , newcomm , error )
IF ( newcomm /= MPI_COMM_NULL ) THEN
    CALL My_collective ( newcomm , ... )
    CALL MPI_Comm_free ( newcomm , error )
END IF
```

# Split (2)

## C example:

```
int colour , error ;  
/* 'colour' is set to an appropriate value */  
MPI_Comm newcomm ;  
  
error = MPI_Comm_split ( MPI_COMM_WORLD ,  
    colour , 0 , & newcomm ) ;  
if ( newcomm != MPI_COMM_NULL ) {  
    My_collective ( newcomm , ... ) ;  
    error = MPI_Comm_free ( newcomm ) ;  
}
```

# More Complex Uses (1)

You can obviously do the above recursively

Change `MPI_COMM_WORLD` to `newcomm`

Change `newcomm` to `evennewercomm`

I said don't **use** overlapping communicators

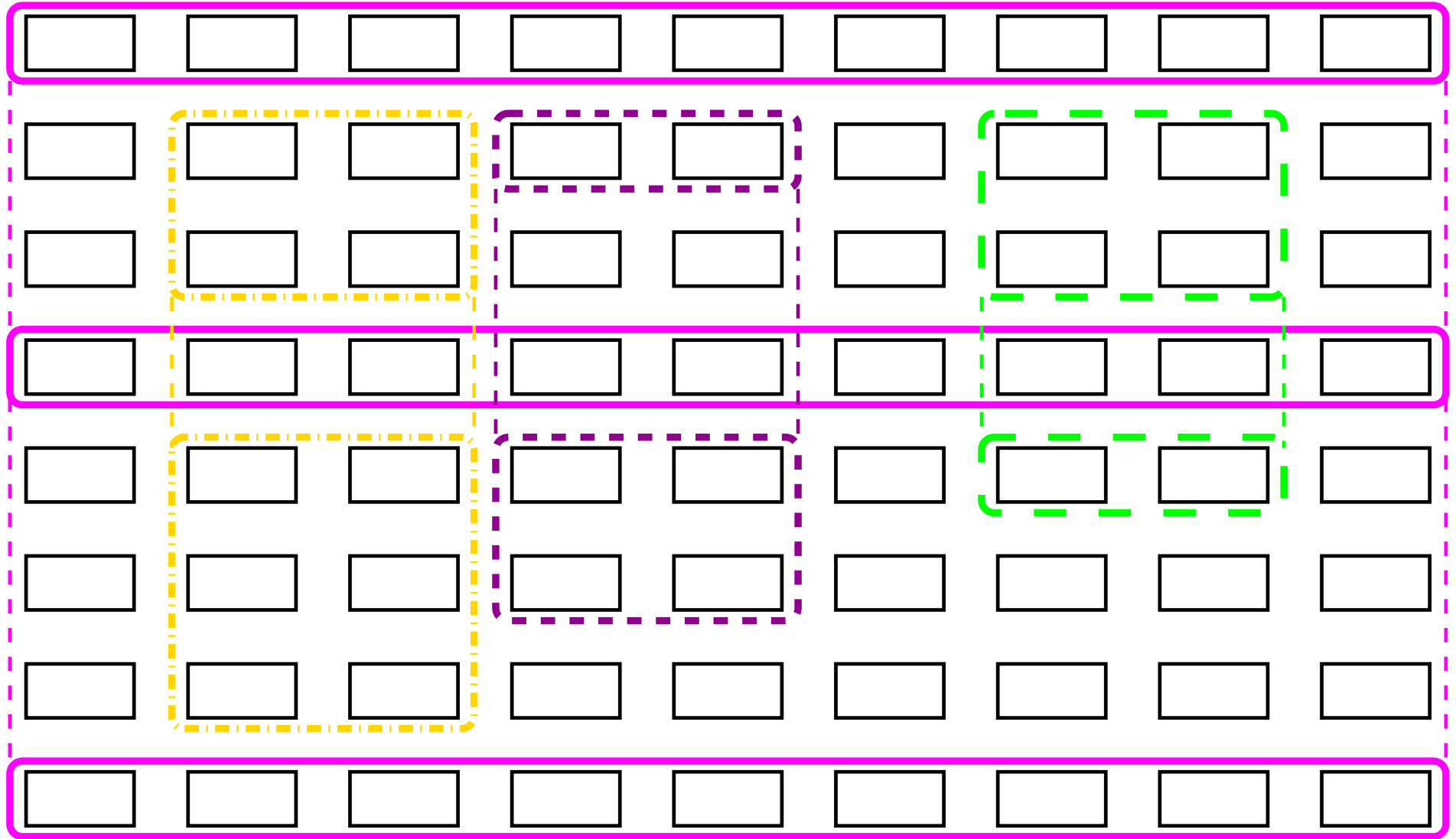
Inactive communicators aren't a problem

- Just tidy up all **transfers** before proceeding  
Suggest using **barriers** for **tuning** reasons

Will give just a very simple, **C**-style example



# Using Two Levels



## More Complex Uses (2)

```
errno = My_global_collective ( MPI_COMM_WORLD ) ;  
errno = Split ( MPI_COMM_WORLD , colour , 0 , & newcomm ) ;  
if ( newcomm != MPI_COMM_NULL )  
    errno = My_split_collective ( newcomm , ... ) ;  
errno = My_global_collective ( MPI_COMM_WORLD ) ;  
if ( newcomm != MPI_COMM_NULL )  
    errno = My_split_collective ( newcomm , ... ) ;  
errno = My_global_collective ( MPI_COMM_WORLD ) ;
```

Note **newcomm** is actually **three** communicators  
They can't **overlap**, so the above use is safe  
Yes, that is **parallel** use of **collectives**

## More Complex Uses (3)

And here is the first half, with some **barriers**  
Probably easier to **tune**, and possibly **faster**  
Note which **communicator** they are used with!

```
errno = My_global_collective ( MPI_COMM_WORLD ) ;  
errno = Split ( MPI_COMM_WORLD , colour , 0 , & newcomm ) ;  
if ( newcomm != MPI_COMM_NULL ) {  
    errno = My_split_collective ( newcomm , ... ) ;  
    errno = Barrier ( newcomm ) ;  
}  
errno = Barrier ( MPI_COMM_WORLD ) ;  
errno = My_global_collective ( MPI_COMM_WORLD ) ;
```

# Error Handling

- The **error handler** is inherited

You can change that subsequently  
I can't imagine many people wanting to

- Remember to set any **error handler** first  
obviously on **MPI\_COMM\_WORLD**

Before creating any **sub-communicators**

# Replication

You can make an exact copy of a **communicator**  
It is then completely independent of the first one  
The function is **MPI\_Comm\_dup**

- Could be useful to bypass **implementation** bugs  
Another possible use is mentioned in extra lectures  
But, in general, very few people will want it

**FFTW** and **SPOOLES** use **MPI\_Comm\_dup**

I think only because they misunderstood MPI  
Possibly to fix up some broken **implementation**

# Topologies

Topologies are how the processes are connected  
MPI's virtual topologies map the program structure

- Independent of the actual hardware network

There is another lecture on Cartesian topologies  
May clarify code that uses an N-dimensional grid

- That use is simple but omitted for brevity

Topologies are almost essential if:

You are writing structure-generic libraries

Your program has a variable graph structure

# Other Facilities

- That's more-or-less all you need to know!

**MPI 2** allowed adding names to **communicators**

Might improve your diagnostics considerably

**MPI\_Comm\_get\_name** & **MPI\_Comm\_set\_name**

One other function, useful for advanced use only

**MPI\_Comm\_compare**

# Groups (1)

There are facilities for operating on **groups**  
Not often used (though I have and **CPMD** does)  
So here is just a **very** brief summary

Operations on **groups** are entirely **local**  
Just operating on sets of integers, after all

For cleanliness, MPI hides them behind a handle  
This is called **MPI\_Group** in **C**  
You should use only the facilities it provides

Take effect only when you create a **communicator**



# Groups (2)

Alternative way of creating **subset communicators**

- **MPI\_Comm\_group** gets the current **group**  
I.e. it extracts it from the **communicator**
- **MPI\_Group\_incl** creates a **subset group**  
You pass it the **ranks** you want to keep
- **MPI\_Comm\_create** makes a new **communicator**  
using the new **subset group**
- **MPI\_Group\_free** releases the **groups**  
Highly desirable to avoid resource leaks
- **MPI\_Comm\_free** is used as earlier

## Groups (3)

**Strongly** advised to program those **collectively**  
I.e. do **identical** group calculations on all **processes**  
Not because MPI needs that – but to avoid errors

Only two actual **collectives**:

**MPI\_Comm\_create** and **MPI\_Comm\_free**

But **group membership** in all **processes** must match

You may find that easier than **MPI\_Comm\_split**

It's purely a matter of **personal preference**

# Other Group Functions

|                        |                           |
|------------------------|---------------------------|
| MPI_Group_compare      | MPI_Group_range_incl      |
| MPI_Group_difference   | MPI_Group_rank            |
| MPI_Group_excl         | MPI_Group_size            |
| MPI_Group_intersection | MPI_Group_translate_ranks |
| MPI_Group_range_excl   | MPI_Group_union           |

Many of them are alternatives to **MPI\_Group\_incl**  
I doubt you will ever want to use the others

# Epilogue

You now know what you can do with **communicators**

Most of you will use only **MPI\_COMM\_WORLD**

One simple exercise using **MPI\_Comm\_split**