# Programming with MPI

*Problem Decomposition*

Nick Maclaren

**nmm1@cam.ac.uk**

March 2010

# Summary

This lecture doesn't teach you what to do

- That is fundamentally problem–dependent

It describes some important possibilities

# Objective (1)

If all processes do the same work, no speedup
A problem must be split between MPI processes

- So the requirement is to divide up the work

Many scientific requirements work on updatable data
E.g. the matrix in Cholesky decomposition
There's often too much for any one of the processes

- So we consider dividing up the data, too

# Objective (2)

- This has nothing to do with MPI, as such

It applies to all distributed memory parallelism
And, to some extent, to shared memory codes

It is usually more difficult than using MPI

- But it is critical to the resulting efficiency

Same thing applies to shared memory parallelism, too
But details of requirements and constraints differ

# Best Approach

Remember: KISS – Keep It Simple and Stupid

- Always start with simplest usable partitioning
- Design your program to allow for change later
- Balance the workload across processes
- Minimise the amount of communication needed
- Gathering data is what reductions are for

# Embarrassingly Parallel (1)

E.g. Monte–Carlo work or parameter searching
Divides naturally into lots of separate tasks

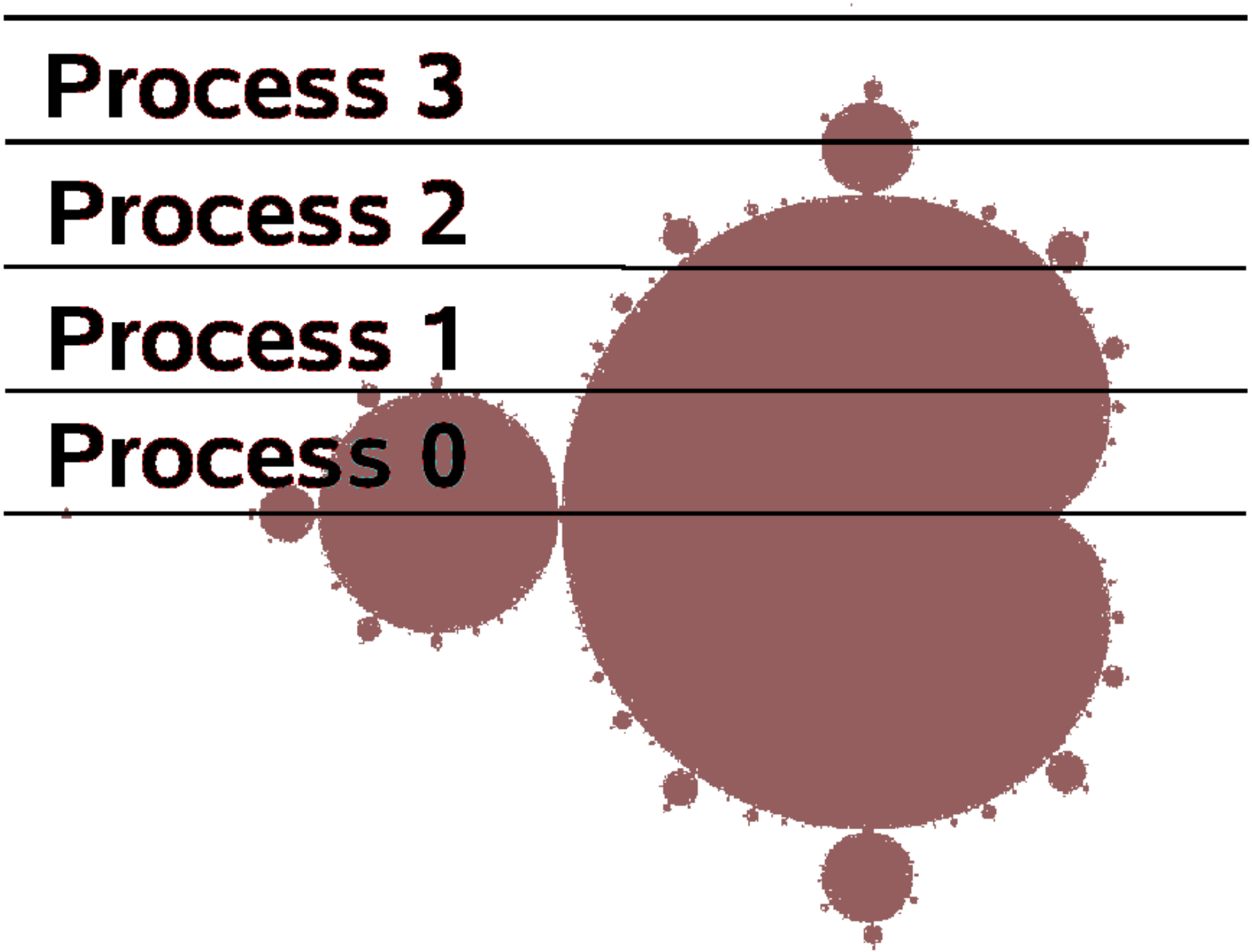
- Each task is largely independent of each other

A master process spawns tasks and collects results
No interaction except during start and termination

- Just divide the tasks between processes

Normally, give all of them an equal number

Very often, that's all you need to do


Process 3
Process 2
Process 1
Process 0

# Embarrassingly Parallel (2)

Remember the Mandelbrot set example?

That divided into sections along the Y axis

- It didn't work very well, as we saw

Problem was correlation of time with partitioning

Randomising points to processes was much better
Can often use a cyclic partitioning instead
Anything that breaks up the correlation

# Embarrassingly Parallel (3)

- Problem is if task time is very variable

Easiest to regard as a statistical distribution

Easy when standard error is smaller than mean
Harder when it is much larger than mean

Relies on the Law of Large Numbers

- Give each process lots of tasks in a run

Preferably much more than (S.E./mean)^2

If you can't, only problem is inefficiency

# Very Nasty Distributions

The Law of Large Numbers does not always hold
Doesn't work if time distribution has no mean
Unfortunately, that really does happen in practice

- Need to be a bit cleverer in that case

Write a simple queue manager

The technique is useful anyway, so worth learning
But won't necessarily deliver high efficiency

- However, if it doesn't, nothing will

# Using Queuing

Master gives each worker process one task to do
Gives it another when it finishes the last

- Master process doesn't do any computation

Can get it to do so, but more advanced use

Writing a queue manager isn't difficult
Good exercise in using MPI_Waitany

- Still a problem if some tasks don't complete

Can introduce statistical and numerical problems

# Partitioning (1)

In general, more communication is involved
Need to decide how to partition the problem

- Try to minimise inter–process communication

Objectives were covered under tuning – remember:

Amount of data transferred between processes
Number of transactions involved in transfer
When one process is waiting for another

- Following slides describe some possibilities

# Partitioning (2)

- Problem may have semi–separate components

E.g. different species in an ecology

- Problem may have a graph structure

In mathematics, that is nodes connected by links

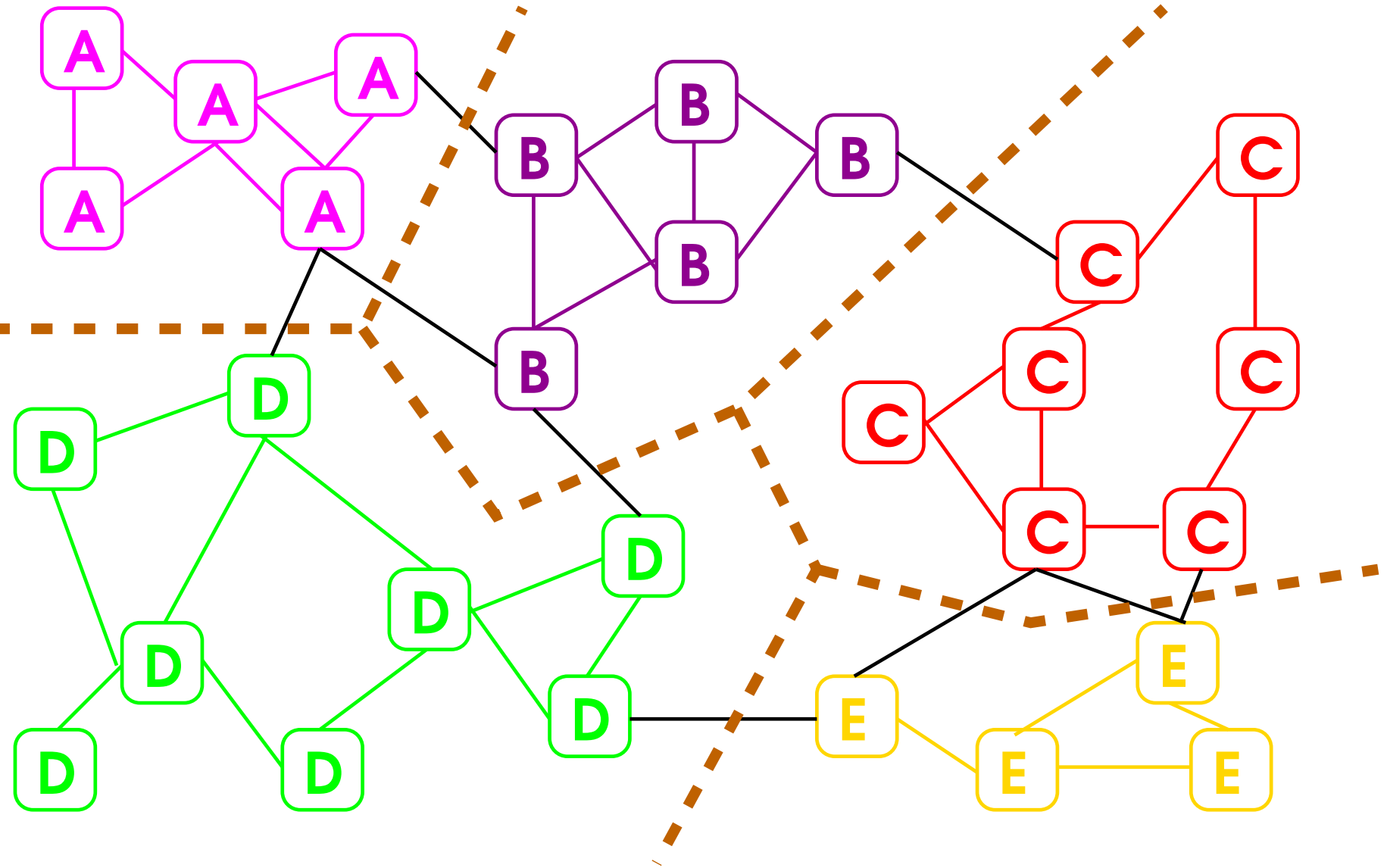
The nodes are the units of data
The links are the communication paths

Generally, look for division that minimises links

- A compromise with balancing workload

$\Rightarrow$ If it works well enough, it's right

# Graph Partitioning



# Rectangular Grids (1)

Problem/data may be in a rectangular grid
Most books and Web pages consider only this one

Regularity means that can analyse properties

- Can sometimes choose best design before coding

Usually easy to parameterise, and tune by experiment

ScaLAPACK puts a lot of effort into supporting this
But it ends up being too complicated and confusing

# Rectangular Grids (2)

- Usually, divide into contiguous blocks

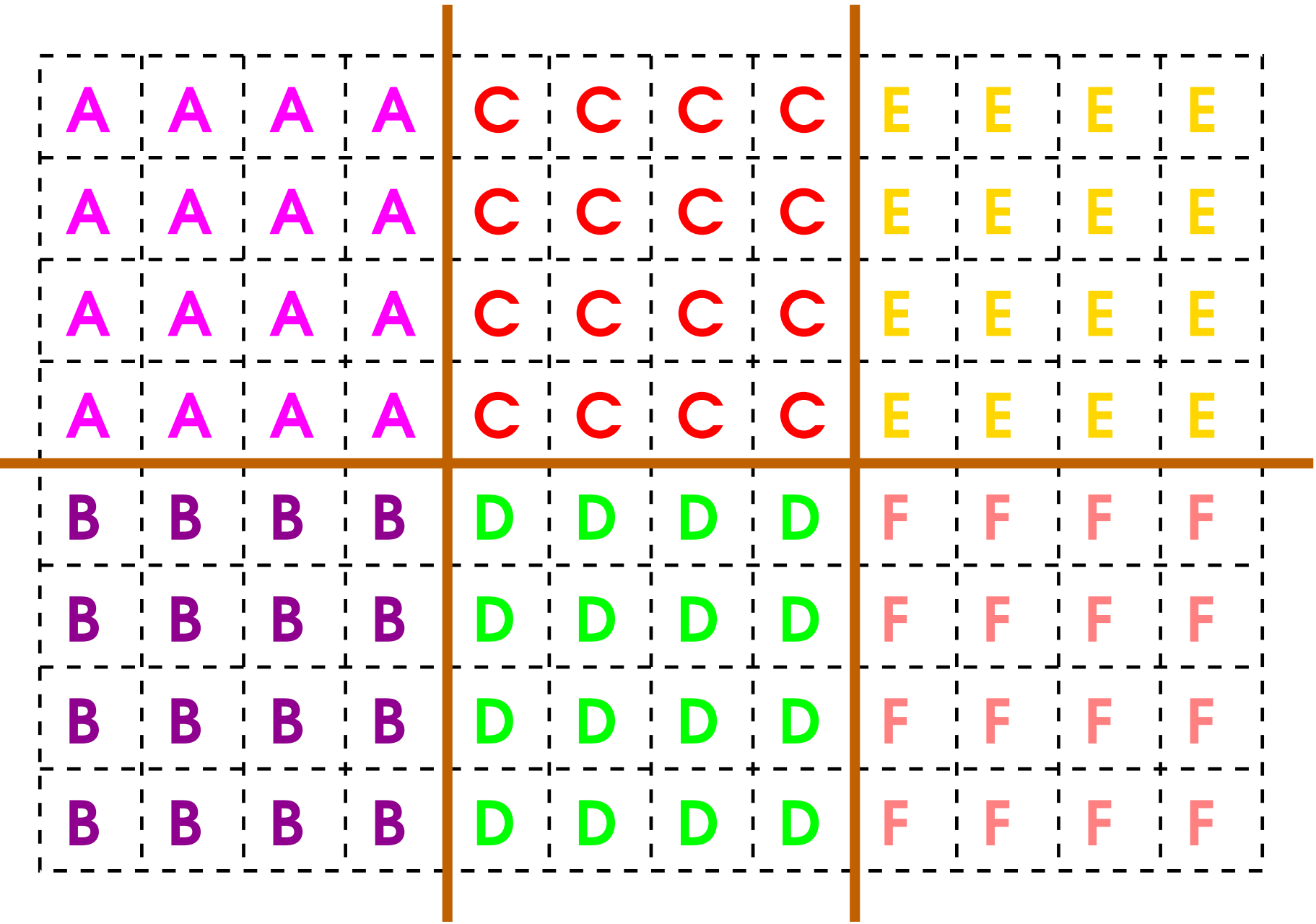
Gives good locality for simple, uniform problems

Best to start here, parameterising block layout
At least the size of blocks, and often shape

- Dividing into strips or layers rarely as efficient

It usually involves more communication

# Block Partitioning



# Rectangular Grids (3)

Sometimes do better with cyclic distribution
There are many variants of cyclic distributions

- Can combine – e.g. cycles of blocks

Or cyclic in one dimension and blocked in another
Or ...

The simplest solution that works is best

# 2–D Cyclic Partitioning (1)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | C | E | A | C | E | A | C | E | A | C | E |
| B | D | F | B | D | F | B | D | F | B | D | F |
| A | C | E | A | C | E | A | C | E | A | C | E |
| B | D | F | B | D | F | B | D | F | B | D | F |
| A | C | E | A | C | E | A | C | E | A | C | E |
| B | D | F | B | D | F | B | D | F | B | D | F |
| A | C | E | A | C | E | A | C | E | A | C | E |
| B | D | F | B | D | F | B | D | F | B | D | F |

# 2–D Cyclic Partitioning (2)

| A | B | C | D | E | F | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|
| F | A | B | C | D | E | F | A | B | C | D | E |
| E | F | A | B | C | D | E | F | A | B | C | D |
| D | E | F | A | B | C | D | E | F | A | B | C |
| C | D | E | F | A | B | C | D | E | F | A | B |
| B | C | D | E | F | A | B | C | D | E | F | A |
| A | B | C | D | E | F | A | B | C | D | E | F |
| F | A | B | C | D | E | F | A | B | C | D | E |

# Non-Uniform Problems

- Problems are very commonly non–uniform

Consider fluid flow around a sharp corner

- Uniform partitioning may not work very well

No option but to balance workload better

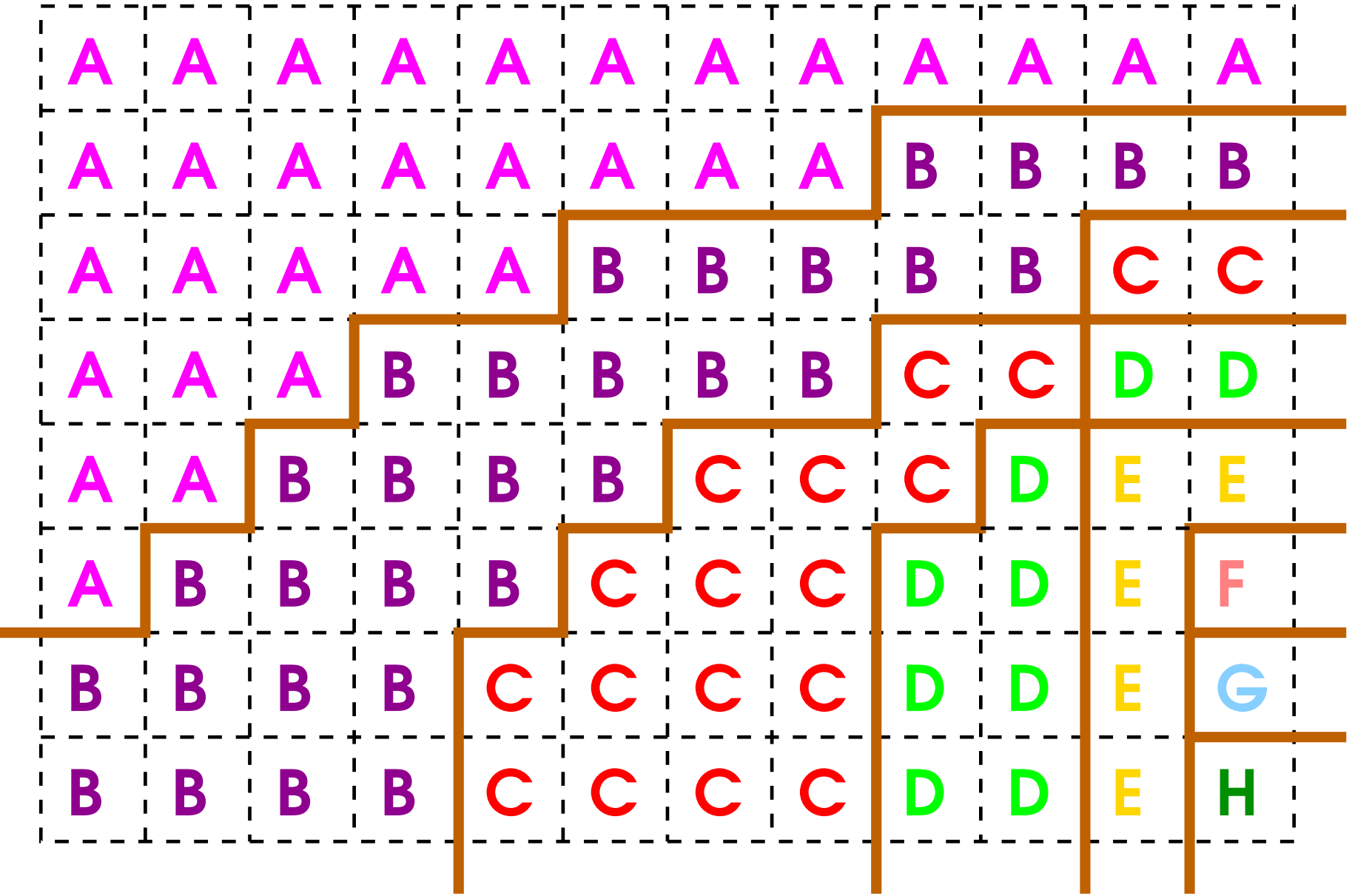
E.g. multi–grid, mesh refinement, coordinate transformation, ...

- More complicated to program and tune

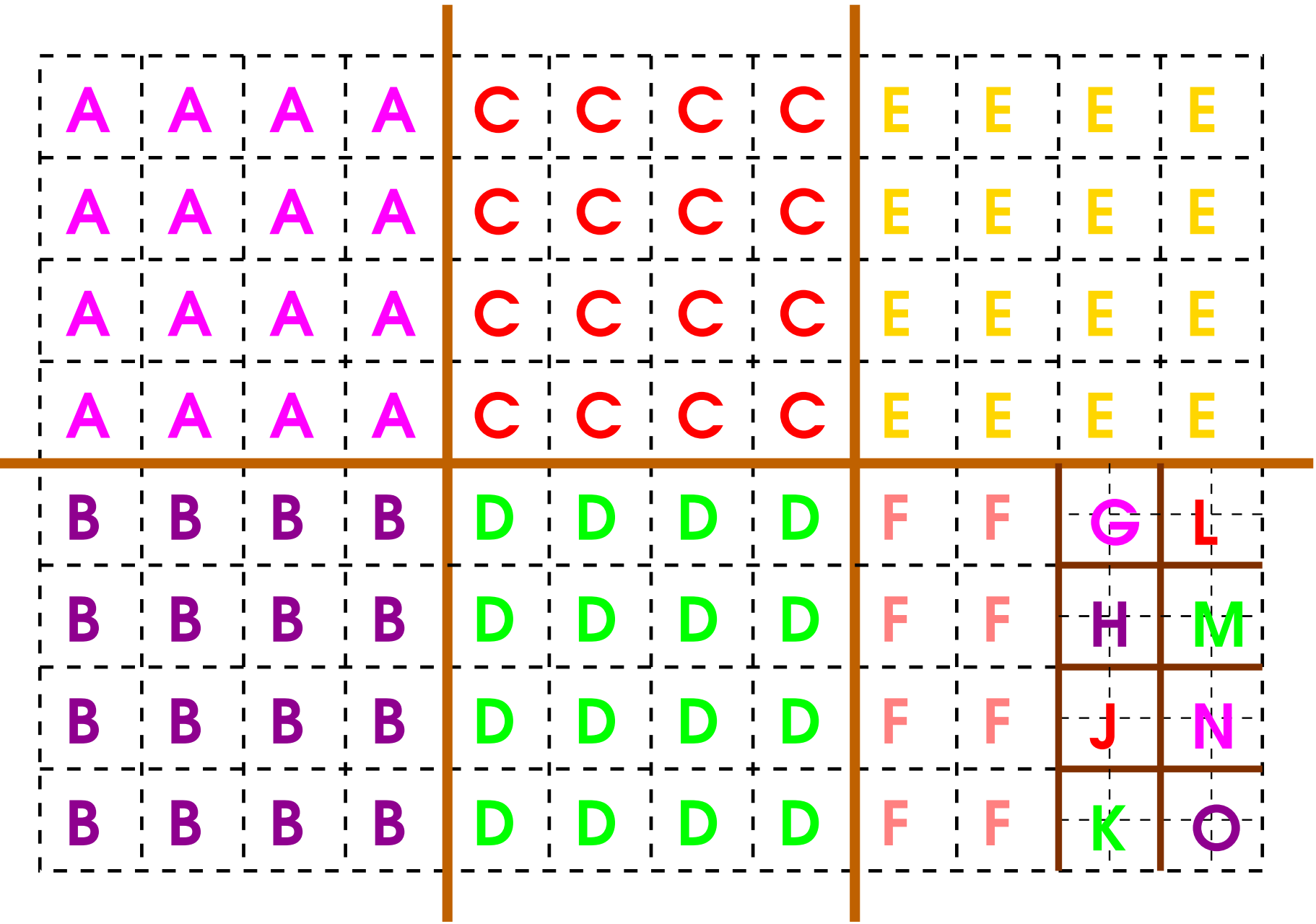
But sometimes gives vastly improved efficiency

Remember, always start with simple partitioning

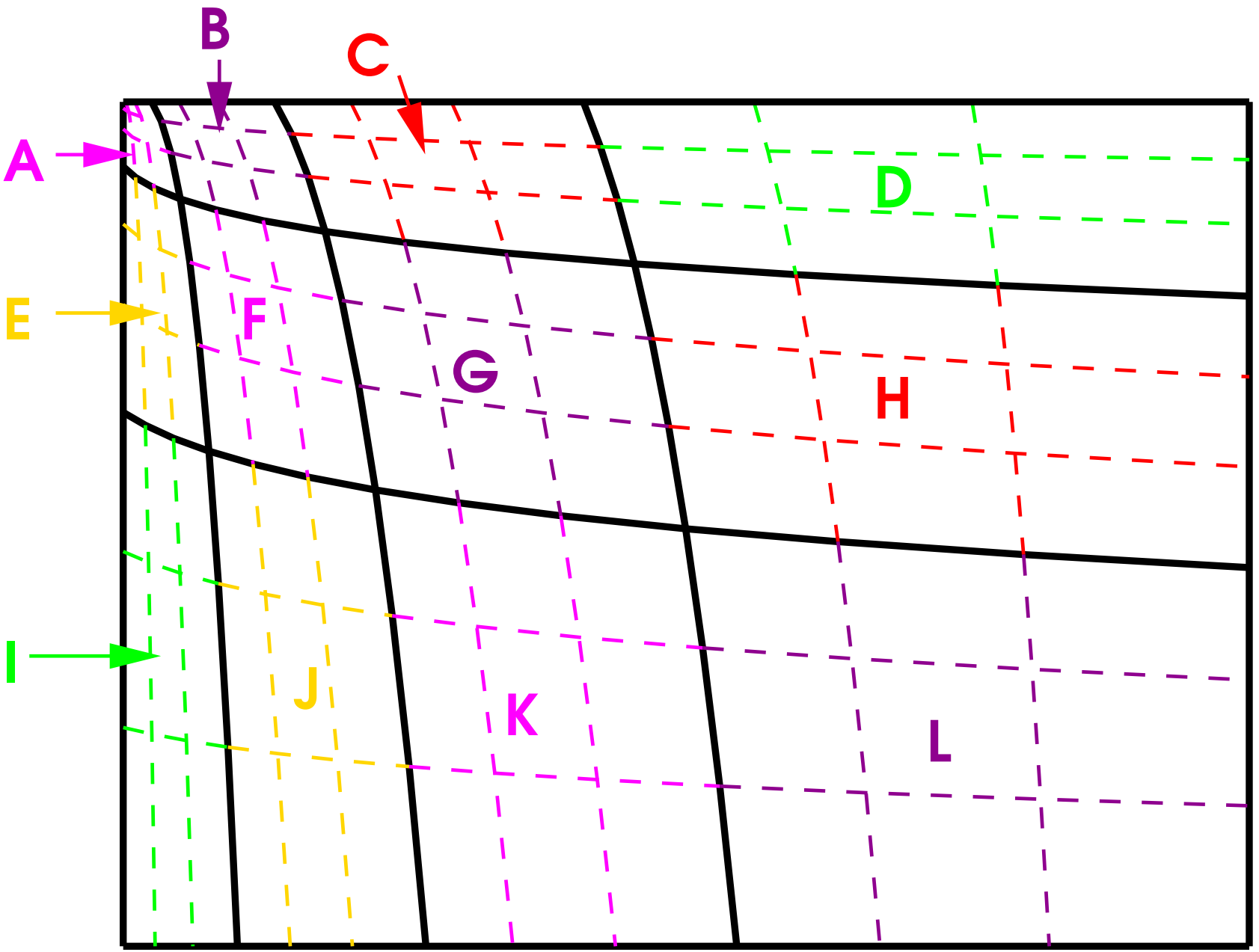
# Irregular Partitioning



# Mesh Refinement



# Transformed Mesh



# Non-Rectangular Grids

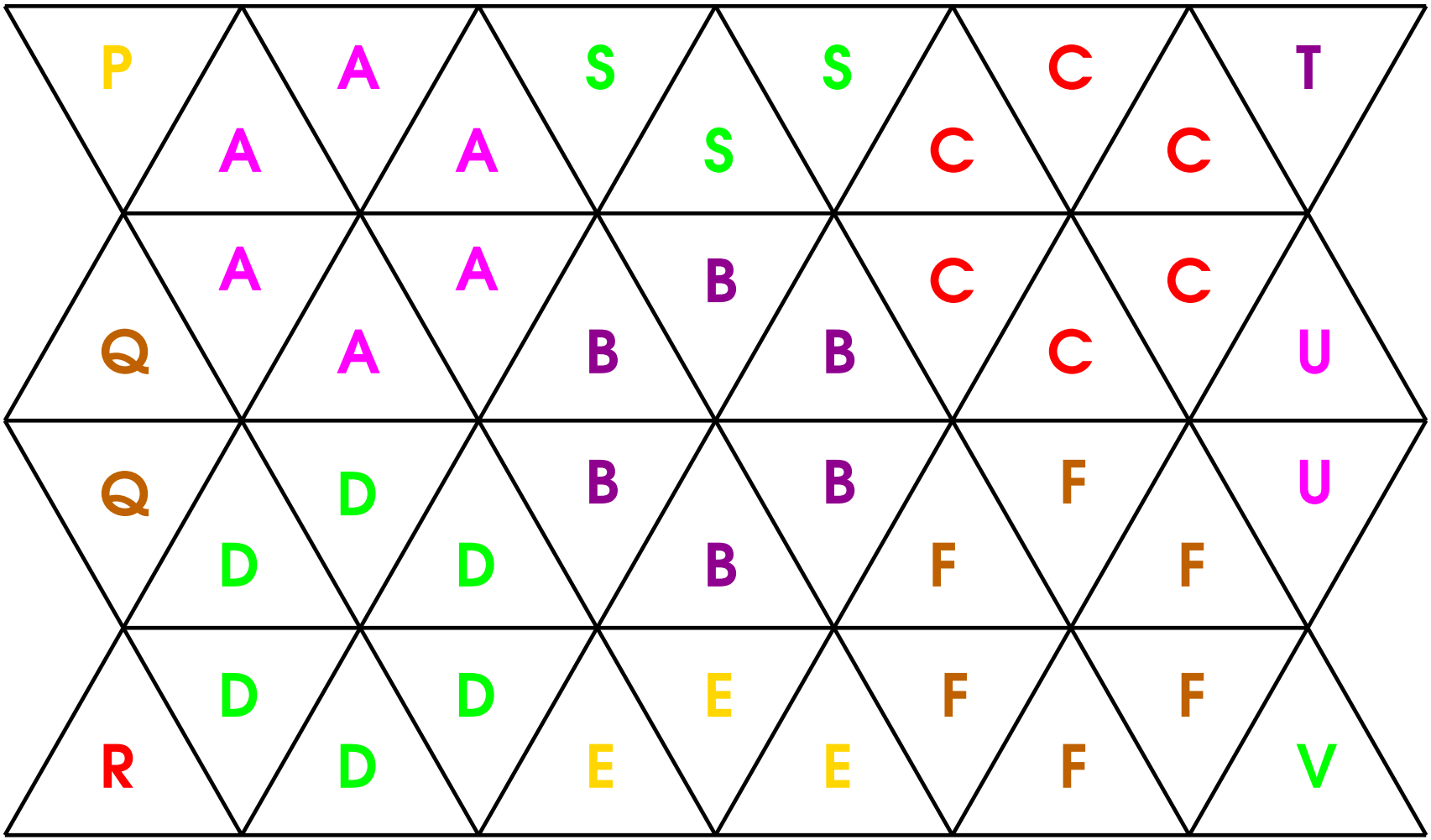
Rectangles are not the only space–filling shapes

Triangles (in 3–D, tetrahedra) are common, too
A common decomposition in finite–element work

- These are regular, but trickier to code

And you may come across other ones – not all regular

# Triangles (or Hexagons)



# Voronoi/Delaunay/Dirichlet

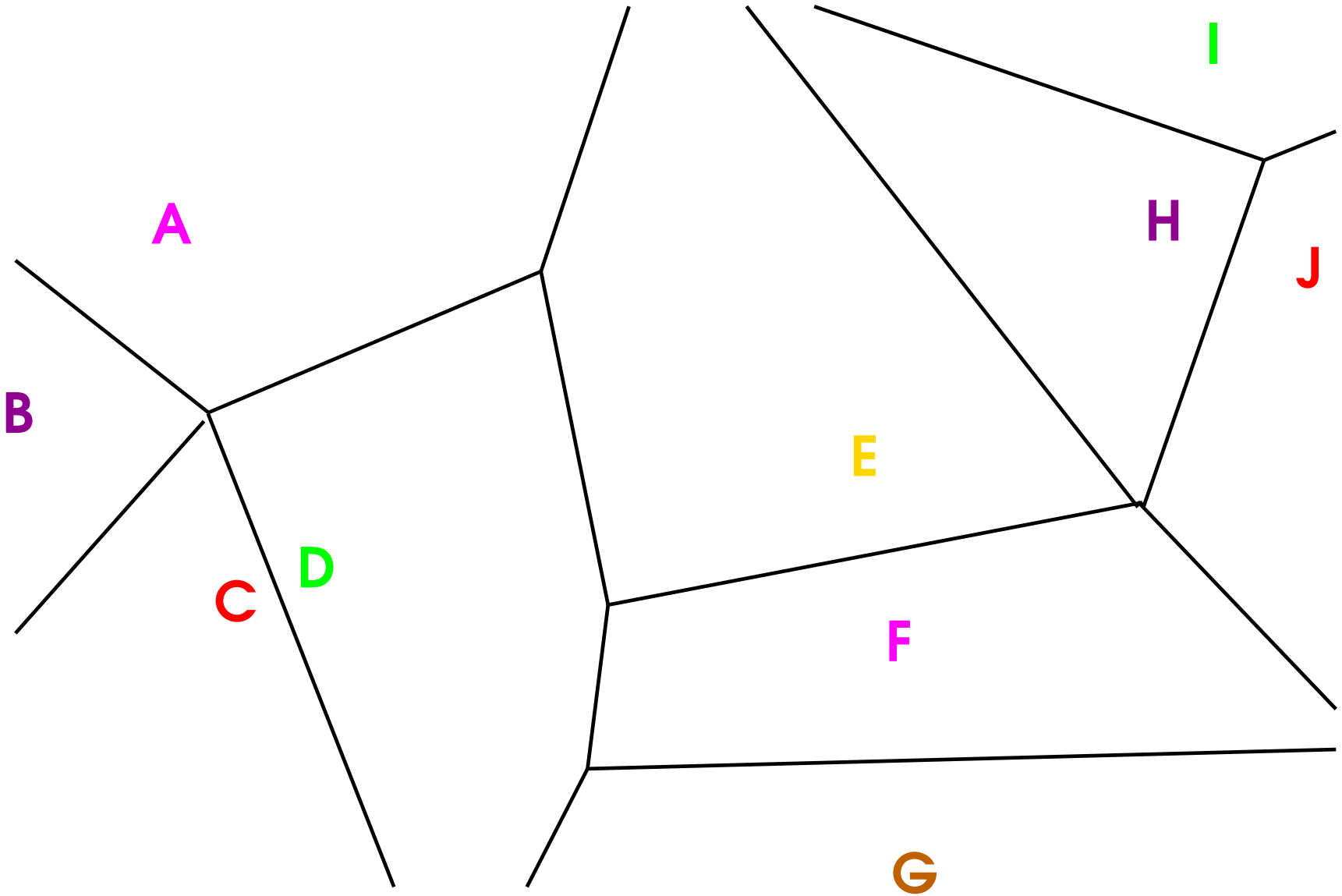
Also Voronoi diagrams, a.k.a. Dirichlet tesselation
a.k.a. Delaunay triangulations etc.
May be taught these in mesh generation lectures

- These are generally used for irregular problems

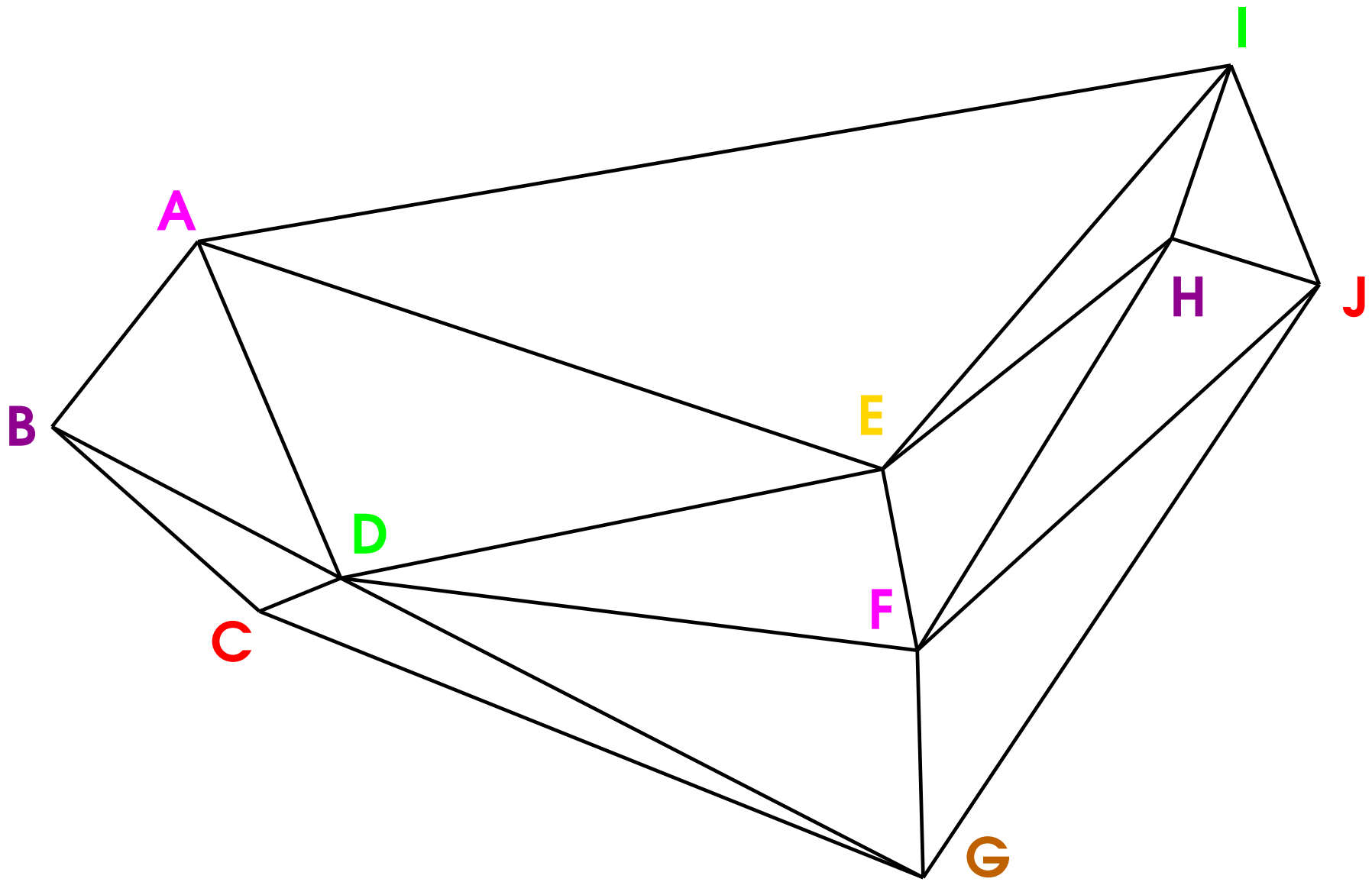
Regard this as a form of graph partitioning

Some very useful mathematical properties
Voronoi is areas nearer to a point than any other
Delaunay has least ill–defined (long, thin) triangles
Used for efficient N–D searching and in other ways

# Voronoi Diagram (a.k.a. Tesselation)



# Delaunay Triangulation



# Communication

Sometimes this is explicit in the problem
E.g. when modelling systems of active components

Very common when problem has graph structure
The links are communication paths

- Just code it, using MPI's facilities

Minimise wait time and amount of data transferred

# Non-Local Data Access

More often in form of non–local data access
Two common variants of this one:

- Direct access, immediately the code needs it

This is often called virtual shared memory

- Division of computation into time–steps

Communicate data in between time–steps

Latter most common use of distributed memory

# Virtual Shared Memory

- You are strongly advised to be cautious

It's extremely hard to design and use correctly

Some designs (e.g. Fortran coarrays) support this
But experts spend a long time designing them

- They will often be built on a basis of MPI

Problem isn't with MPI, but design and discipline

Use those designs, but don't invent your own

# Time-Step Designs

Very common for things like PDEs and ODEs
So you you may well be using them anyway

You can then resynchronise data each time–step
Can do it either by reading or writing:

- Broadcast local data between each time–step

Use others' previous data during next time–step

- Create updates to other processes' data

Send them to other process between each time–step
Less commonly used – and not covered further

# General Approach

Each process owns a subset of the global data

- It updates it – other processes only read it

In the simplest case:

All processes broadcast all data between steps
Complete, global read access during next time–steps

- Good if long time–steps, and amount of data small

May be too much data, or broadcast cost too high

Advantage is very easy to design and use correctly

# Boundary Data Sharing (1)

Can distribute only data that will be needed

- Typically the data near the boundary of processes

PDEs obvious example – need only nearby data

Obviously more complicated to design and program
But can reduce amount of memory a great deal
And reduce communication cost even more

- Keep your design simple, and code it carefully

Don't worry about minor inefficiencies

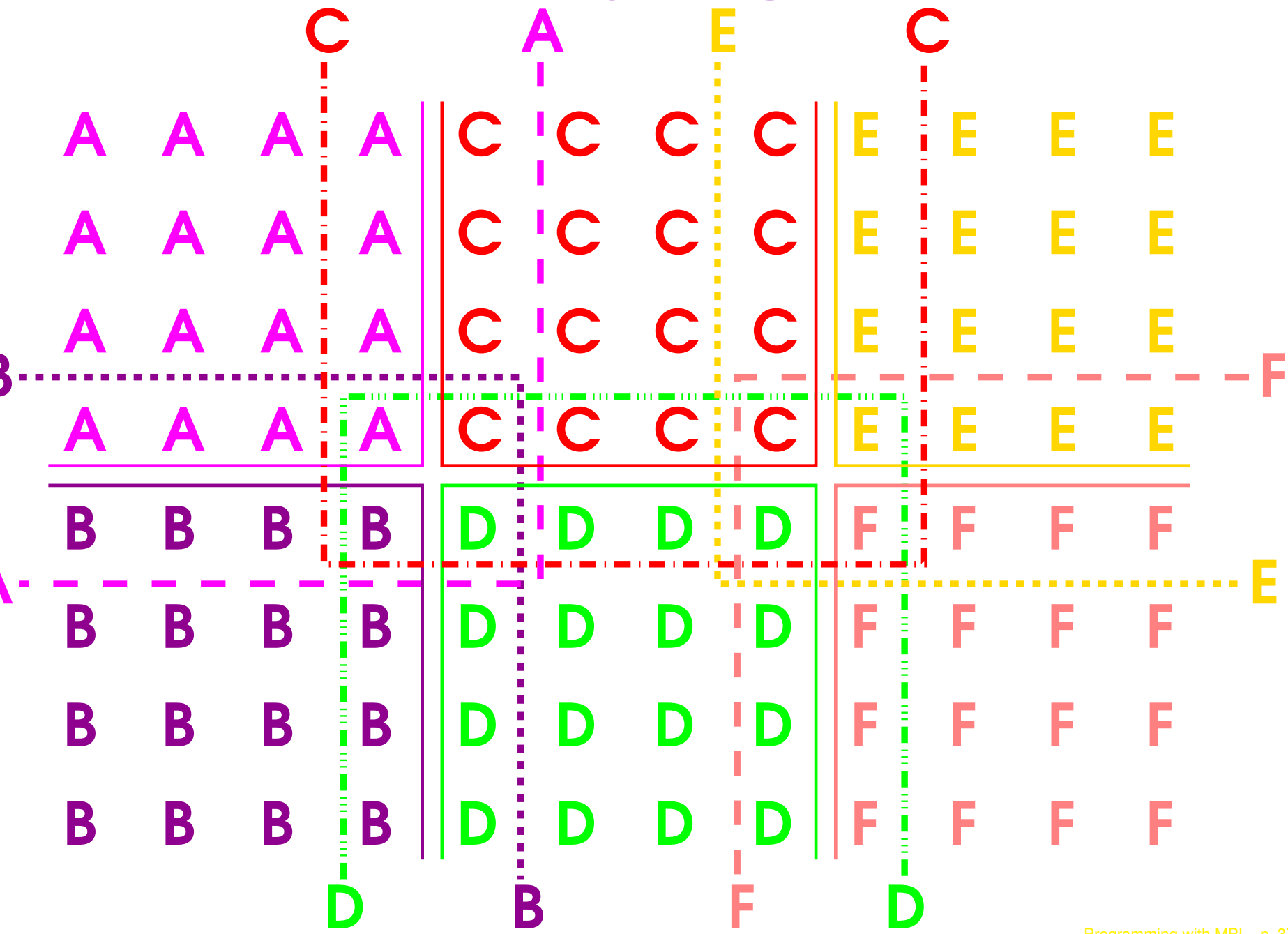
# Boundary Data Sharing (2)

- Each process stores its own data plus boundaries

I.e. surrounding data owned by other processes

- Updates only its own data, not boundaries
- Gets updated boundaries from other processes
- Sends its own edge data to nearby processes

# Boundary Regions



# Scalability Warning

Generally assumes boundaries are very thin
If this is not so, better to broadcast all data
Two guidelines for when to do that:

- Most of the data is in some node's boundary

You probably won't gain any performance

- Boundaries go beyond the immediate neighbours

That's fiendishly hard to program correctly
Multi–cell boundaries are no problem, however

# Repartitioning

Part of program needs one partition design
And other parts need other partition designs

- You can repartition between those parts

Obviously worthwhile only if parts are heavyweight
Repartitioning all data can be extremely slow

- Key is to keep parts of program very separate

Almost like different programs merged into one

- Generally, not worthwhile on simple programs

# Reblocking

Probably the simplest form of repartitioning
Takes one blocked design and converts to another

- Think of matrix transpose – e.g. MPI_Alltoall

Commonly used to implement n–D FFTs efficiently

For those, FFT algorithm is used on one dimension
It applies that to the others as vector data

Most efficient if processes divide up the vectors
So reblock/transpose between each dimension's FFT

# An Extra Lecture

MPI has some management facilities for this
Not covered, but there are slides and practicals
Worth using for parameterised decompositions

Topologies
Allow managing n–D indexing more generically

MPI also has facilities for graph decompositions
Definitely complicated, but graph decomposition is

# Reminder

- Partitioning is key to efficiency in many problems
- Don't rush in – design it carefully
- Choose the one that best matches your problem
- KISS – Keep It Simple and Stupid

# Practicals

There are some heavyweight practicals
Very similar to using MPI "for real"

The first is master/worker controller code
Useful for embarrassingly parallel problems

- It is much easier than it looks

Second is a grid decomposition problem
Like most code for PDEs, finite elements etc.

- It is much harder than it looks

$\Rightarrow$ Don't bother now – you don't have time

# Omitted Lectures

Some material in previous lectures (like progress)

Error Handling – for diagnostics and tidying up

Communicators etc. – using subsets of processes

More on Point–to–Point
Mainly non–blocking (asynchronous) transfers

Topologies – managing n–D indexing

The Extra Lectures – mainly more detail

# Extra Lectures (1)

- Follow the guidelines here and rarely need them

Worth scanning later, for use with production code

MPI/

Miscellaneous Guidelines contains extracts

It covers what you absolutely must know

More details are in the following three lectures:

# Extra Lectures (2)

## Composite Types and Language Standards

It's mainly more on what not to do
But avoiding the "gotchas" is very important

## Debugging, Performance and Tuning

A lot of things that you don't want to know
But which you may need to, if you are unlucky

## Attributes and I/O

It's mainly going though I/O handling in detail

# Extra Lectures (3)

You probably won't want to look at any of these

One–sided communication
Absolute basics about MPI's RDMA support

Advanced Completion Issues
About point–to–point usage I don't recommend

Other Features Not Covered
What the course doesn't cover and why