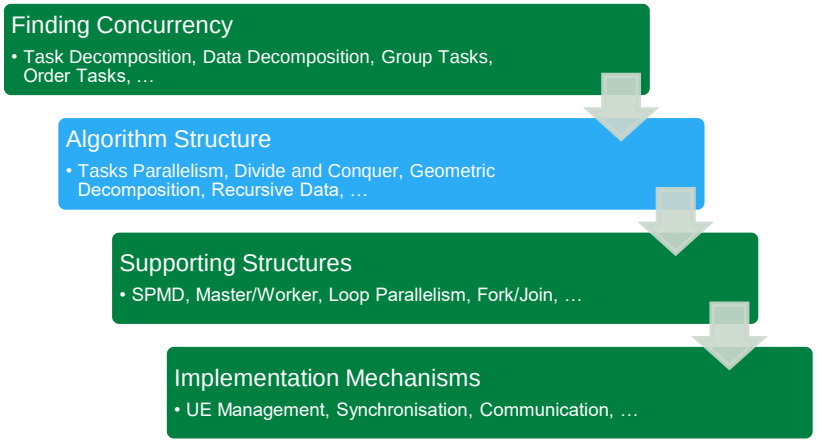
Parallel Design Patterns-L03

Geometric Decomposition,

Recursive data

Course Organiser: Dr Nick Brown
nick.brown@ed.ac.uk
Bayes room 2.13



Strategy:

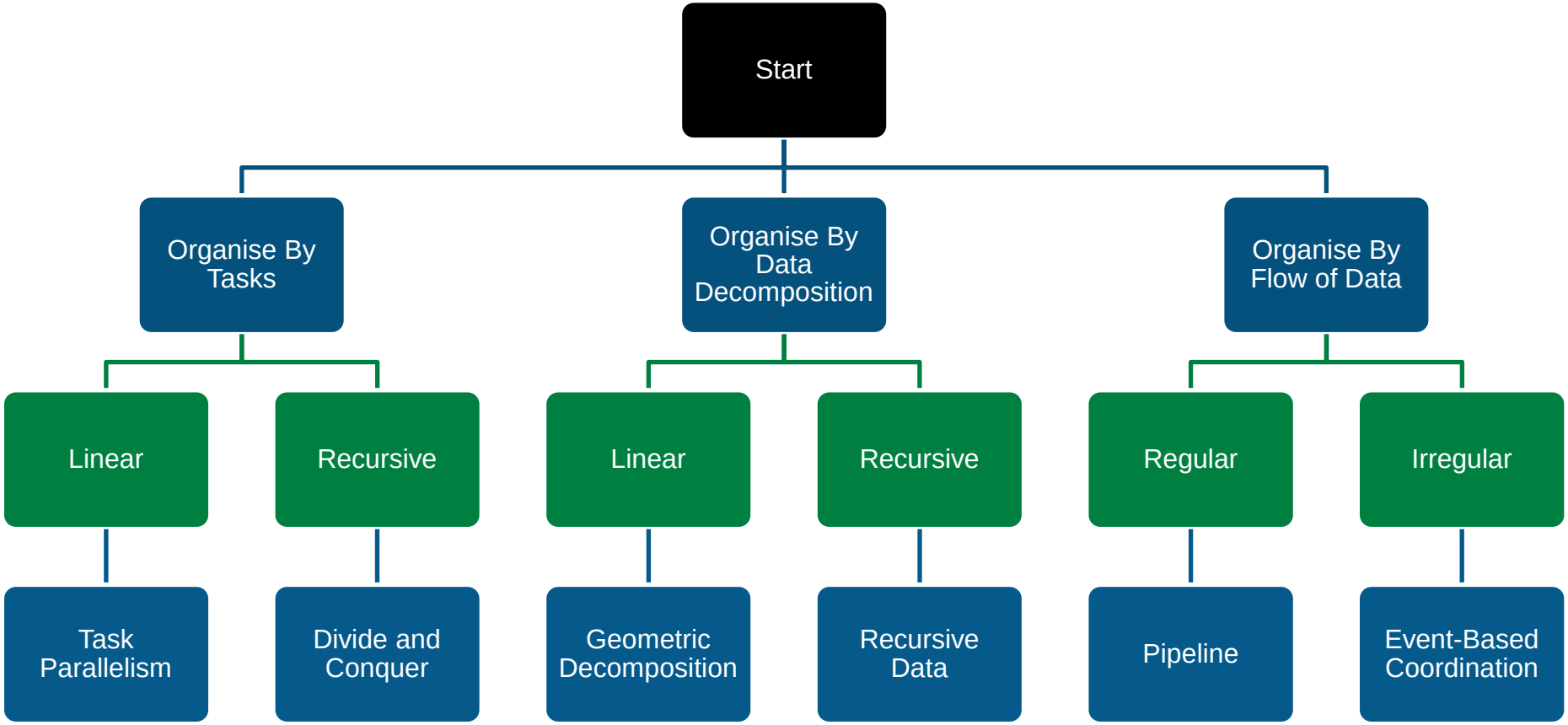
PARALLEL ALGORITHM STRATEGY

Strategy Choice: Things to keep in mind

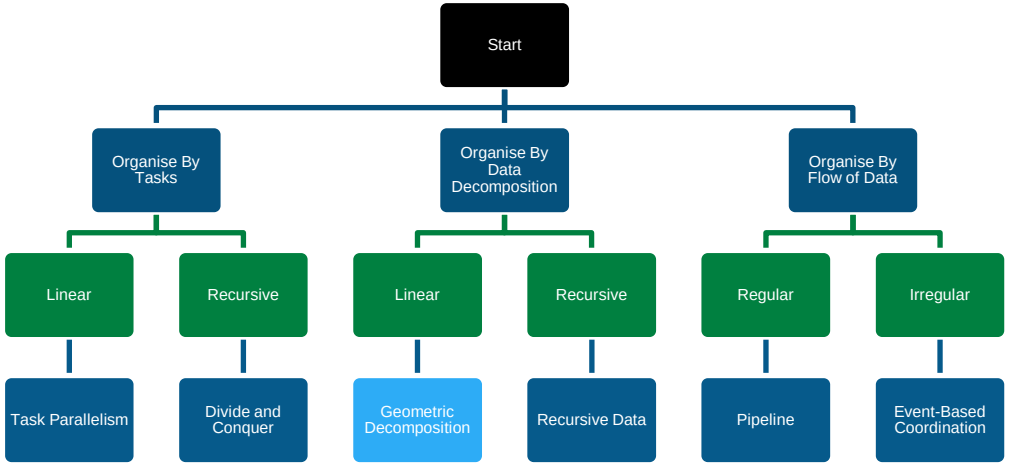
- Efficiency
  - Speed, memory, storage
- Scalability
  - Large machines, large problems
- Simplicity of the code
  - Development, debugging, verification, modification, maintenance
- Portability
  - Software nearly always outlives its original target platform

- There is rarely *one right answer* and a good design often boils down to a number of *tradeoffs*
- Often useful to think *Structure First, Optimisation Second*

Parallel Algorithm Strategy



After ***Patterns for Parallel Programming***: Mattson, Sanders, Massingill; Addison Wesley (2005)



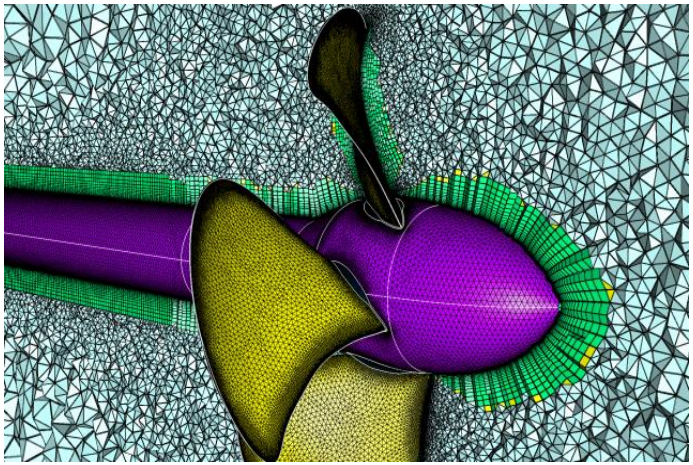
Pattern:

GEOMETRIC DECOMPOSITION

Also known as domain decomposition or course grained data parallelism

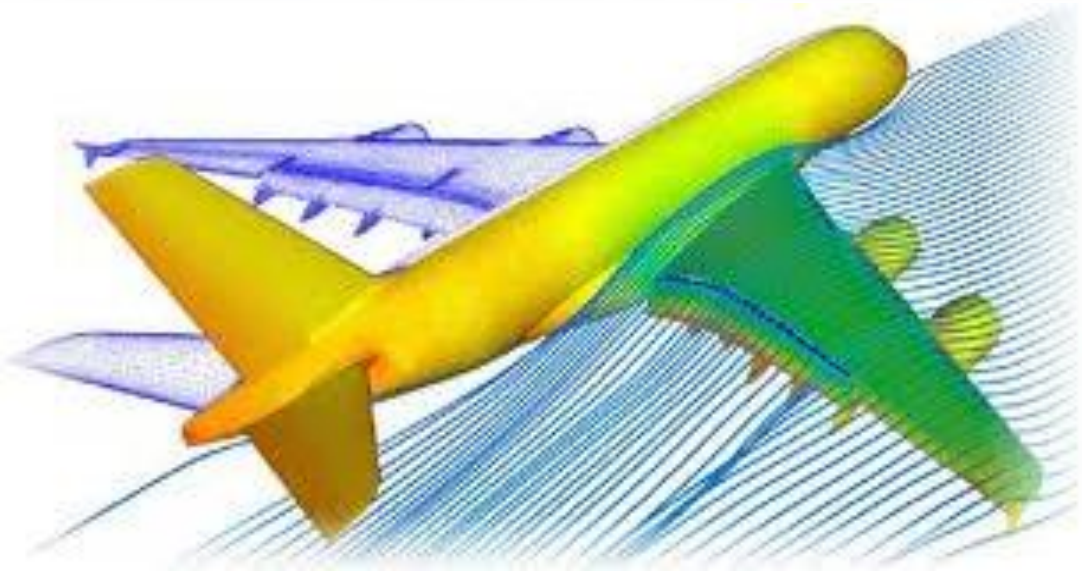
Geometric Decomposition – Problem

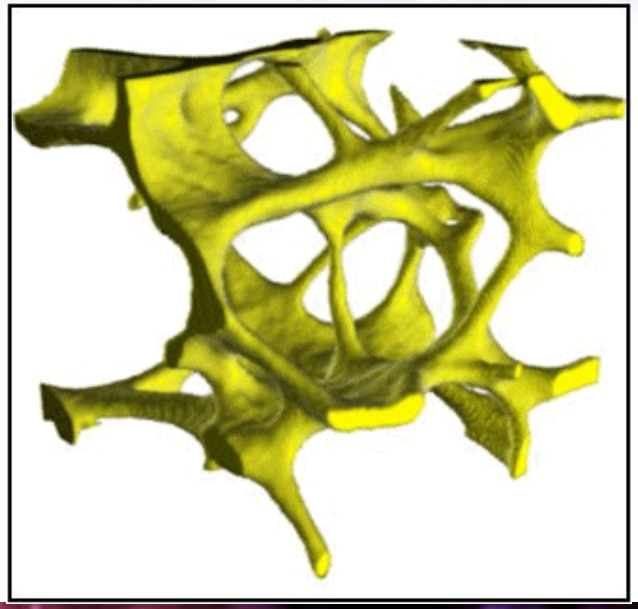
- Given a problem which can be broken up into data areas that can be operated on concurrently, how can an algorithm be organized so as to exploit this potential parallelism?

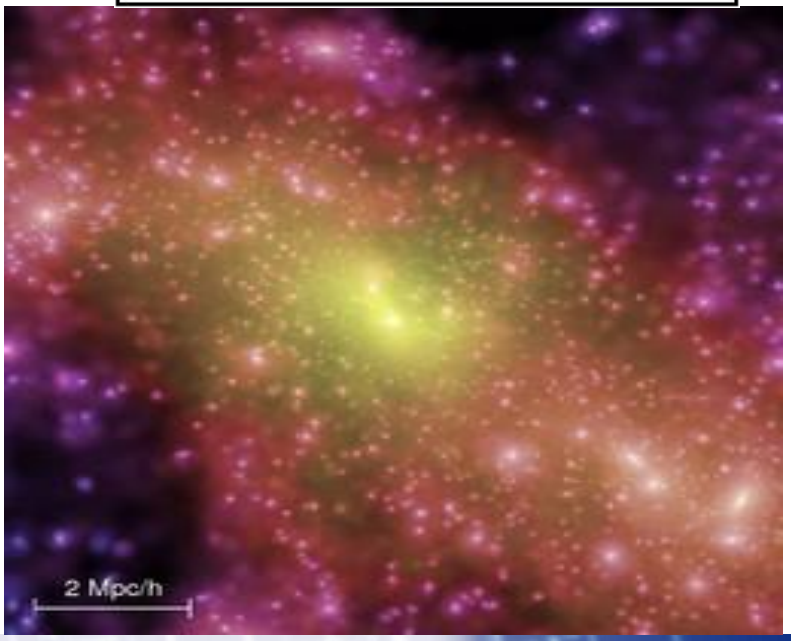


- The algorithm will probably involve one key data structure whose elements can be operated on concurrently
  - Data structure often an array, but could also be a graph
  - Not ideal for data structures with inherent hierarchy such as trees which are often better dealt with by the *recursive data* pattern
- Operations on an element typically involve the element itself and some neighbouring elements

Some examples of use







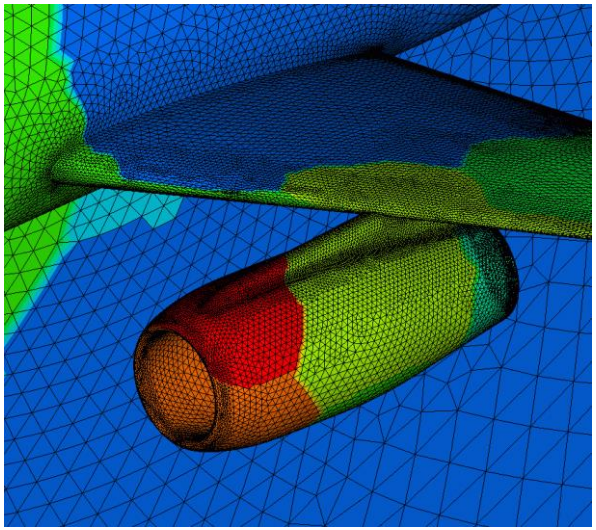
Geometric Decomposition - Forces

- How to define subdomains and assign these to units of execution (UEs.)
- Considering…
  - Efficiency
  - Simplicity
  - Portability
  - Scalability

- Again: An important consideration here is **load balance**
- Ensuring required data is available to perform the operation on the subdomain
- All decomposition approaches introduce parallelisation overheads

Geometric Decomposition – Solution

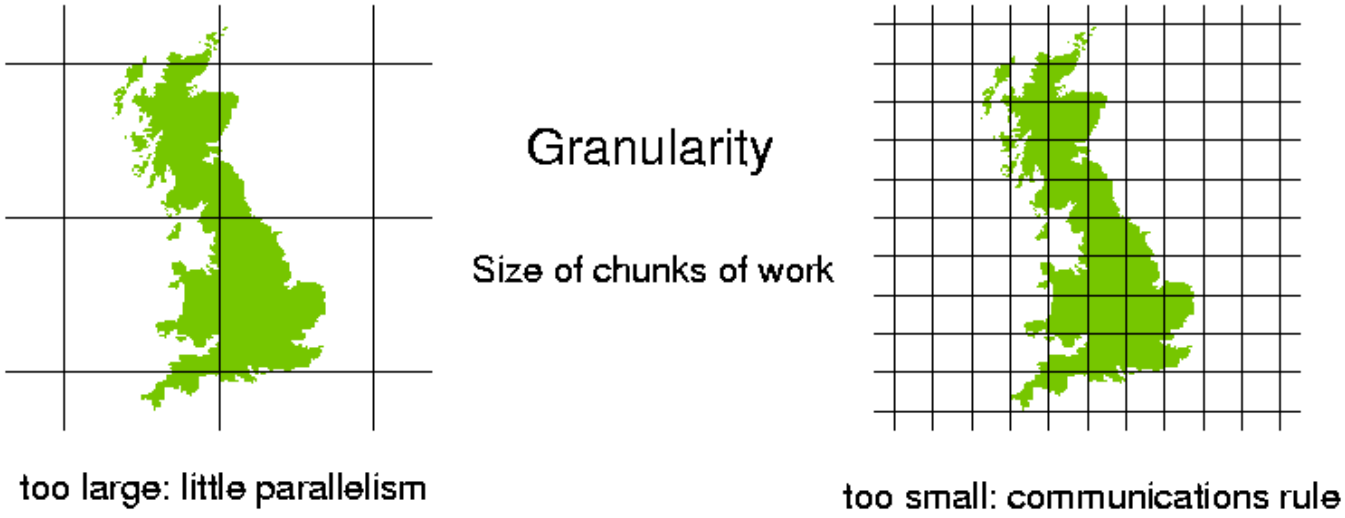
- Consider each of the following:
  1. Data decomposition
     - How to split up the domain into subdomains
  2. Exchange operation
     - Influence of neighbouring subdomains
  3. Update operation
     - The computational work…
  4. Task scheduling
  5. Program structure



Data Decomposition

- Shape of subdomains
  - Regular / Irregular
  - Decompose in how many dimensions?
  - Answer often depends on symmetries in underlying problem and potentially on the target hardware
- Where does data start off?
  - Already organised in terms of domain on disk?
  - Can read or generate initial state in parallel, or read in from one UE and then distribute
- Will this result in an even or uneven distribution of data?
- Granularity of data decomposition is important
  - Important for efficiency
  - Related to balance between communication and computation

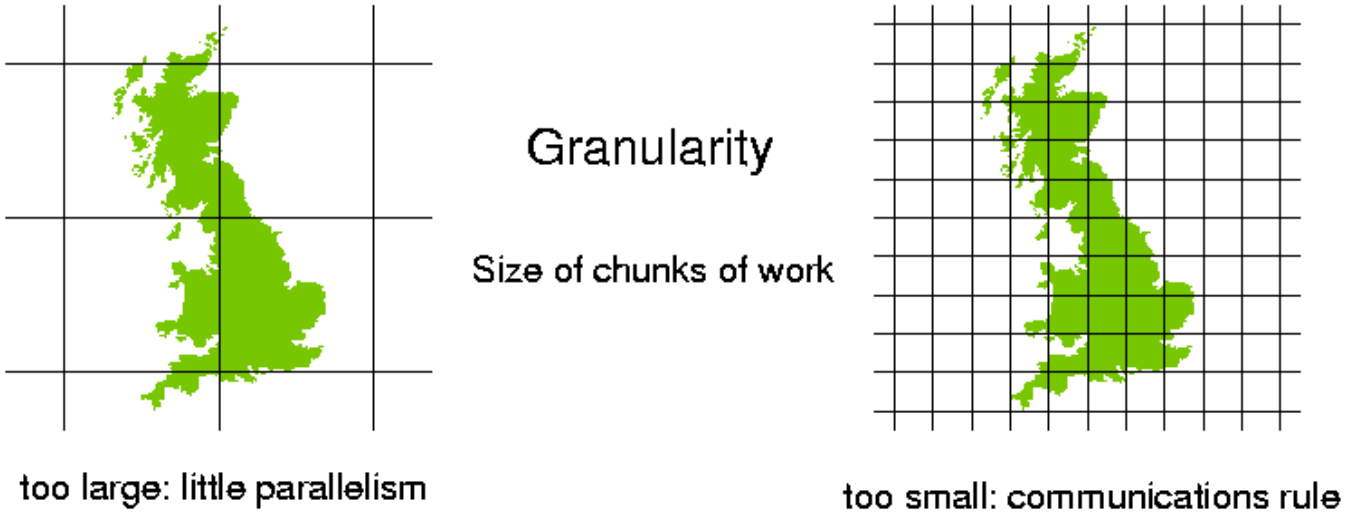
Granularity



- Coarse-grained = Small number of large subdomains
- Fine-grained = Large number of small subdomains
  - Having more subdomains increases communication

- Splitting a problem has time cost, but this can be recouped through parallelism

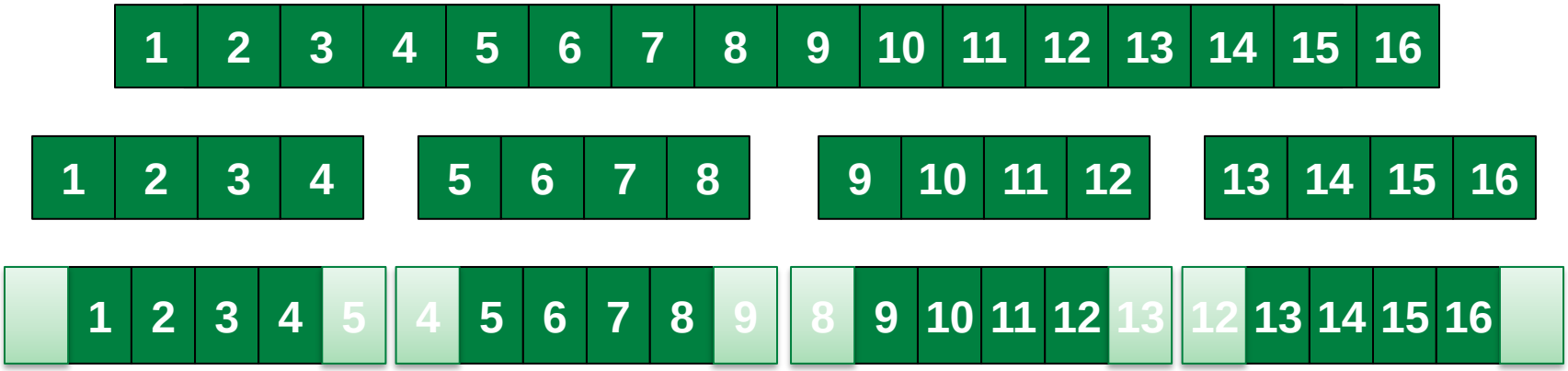
Granularity



- Usually not possible to determine optimum granularity theoretically
  - Need to measure experimentally
  - Will probably depend on problem size and target architecture (especially the relative strengths of processing and communications network) so this should probably be user-tuneable at compile or runtime

The Exchange Interaction: Halo Swaps

- Sub-domains need to have knowledge of their neighbours in order to perform their part of the communication
- Need to ensure that non-local data is present before it is used
- Common approach is to use *halo-swapping*
  - a.k.a. *ghost* or *shadow* boundaries



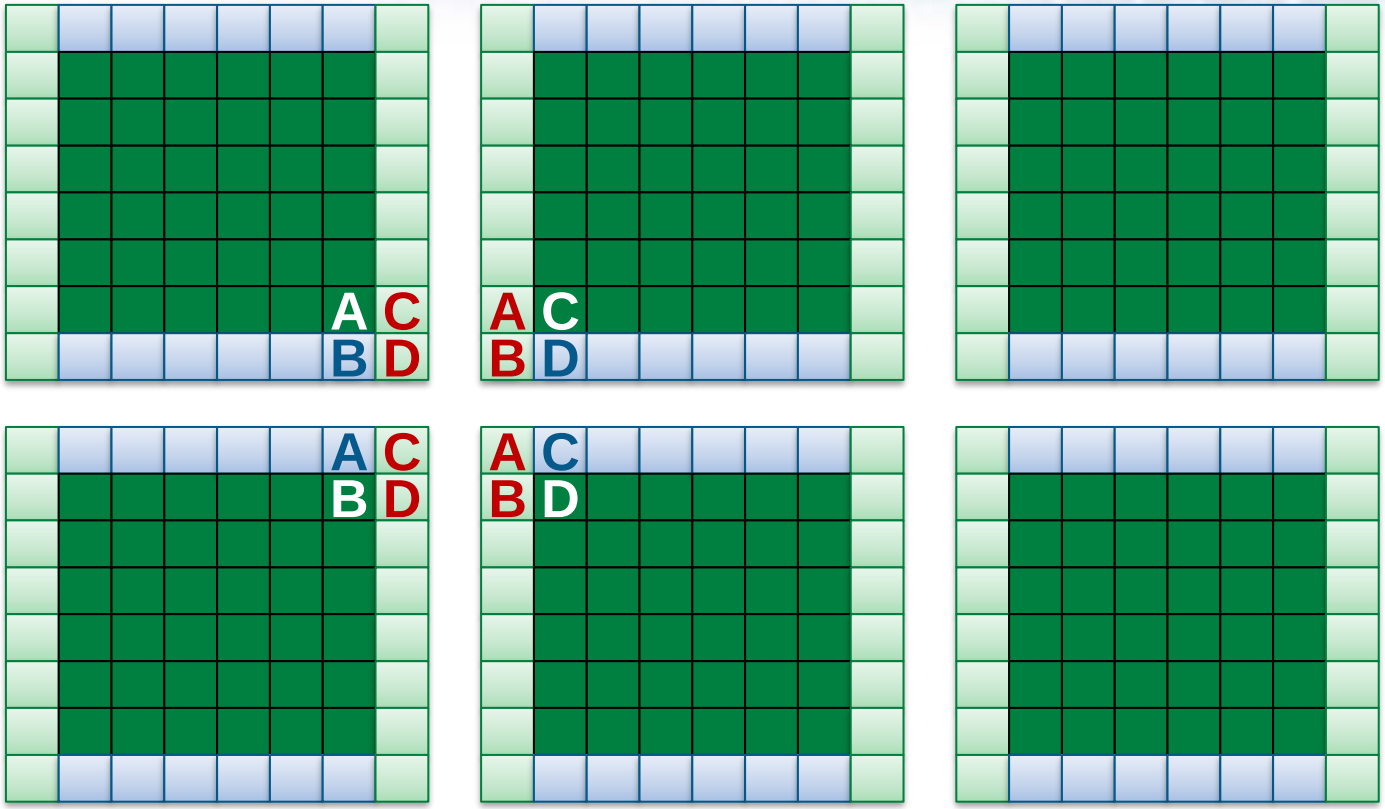
2D Geometric Decomposition

- Simple case:

```
Loop
    Update halos
    Perform calculation
End loop
```

- Efficiencies can be made by grouping together the communications associated with swapping each side of a halo
- All decomposition approaches introduce overheads
  - transferring data on the boundaries
  - synchronisation
  - calculating global quantities
  - Volume gives us computation, surface area communication

Example matching halos to elements



- Halos usually swapped after every time step which implies synchronisation
- The depth of the stencil is the number of neighbouring elements required in a direction (here it is 1.)

The update (computation) operation

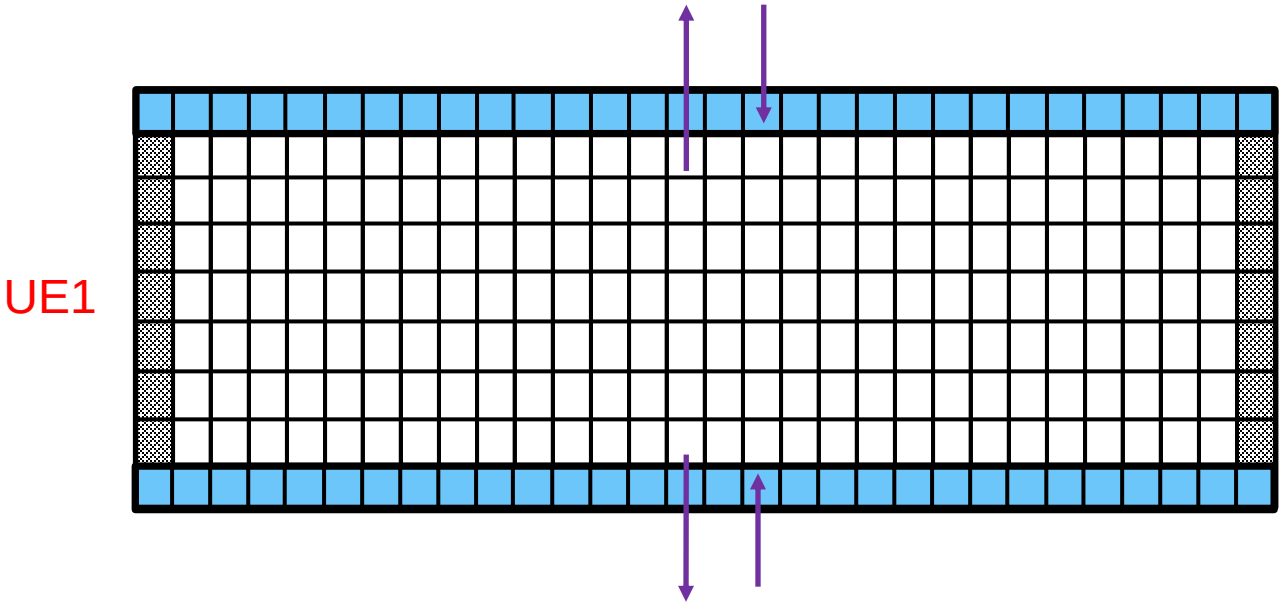
- Conceptually simplest to complete the exchange before starting the update

```
Loop
    Update halos
    Perform calculation
End loop
```

- In practice, better performance can be obtained by overlapping communication and computation
- …which can be implemented in various ways, e.g.,
  - Multithreading within a task
  - Non-blocking MPI communications
- Need to ensure that correct neighbouring data is present before performing the update operation

Overlapping compute and communication

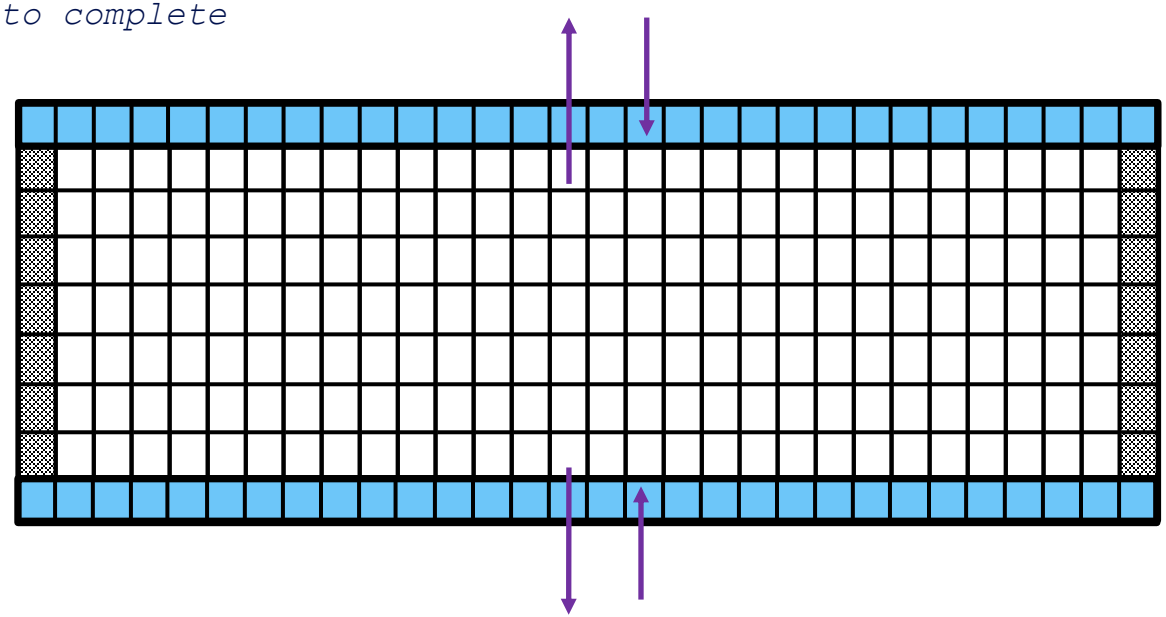
```
……
for (k=0;k<MAX_ITERATIONS;k++) {
    non blocking halo swaps
    block for all communications to complete
    for (j=1;j<=local_nx;j++) {
        for (i=1;i<=ny;i++) {
            tmpnorm=tmpnorm+pow(u_k[i+(j*mem_size_y)]*4-
                u_k[(i-1)+(j*mem_size_y)]-u_k[(i+1)+(j*mem_size_y)]-
                u_k[i+((j-1)*mem_size_y)]-u_k[i+((j+1)*mem_size_y)], 2);
        }
    }
    ……
}
……
```



Overlapping compute and communication

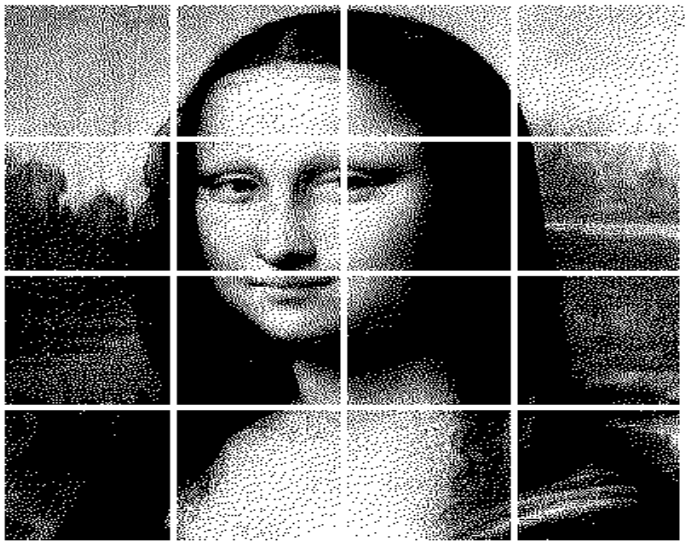
```
……
for (k=0;k<MAX_ITERATIONS;k++) {
    non blocking halo swaps
    for (j=2;j<local_nx;j++) {
        for (i=1;i<=ny;i++) {
           tmpnorm=tmpnorm+pow(u_k[i+(j*mem_size_y)]*4-
                 u_k[(i-1)+(j*mem_size_y)]-u_k[(i+1)+(j*mem_size_y)]-
                 u_k[i+((j-1)*mem_size_y)]-u_k[i+((j+1)*mem_size_y)], 2);
        }
    }
    block for all communications to complete
    for (i=1;i<=ny;i++) {
          tmpnorm=tmpnorm+ ……
    }
    for (i=1;i<=ny;i++) {
          tmpnorm=tmpnorm+……
    }
    ……
}
```

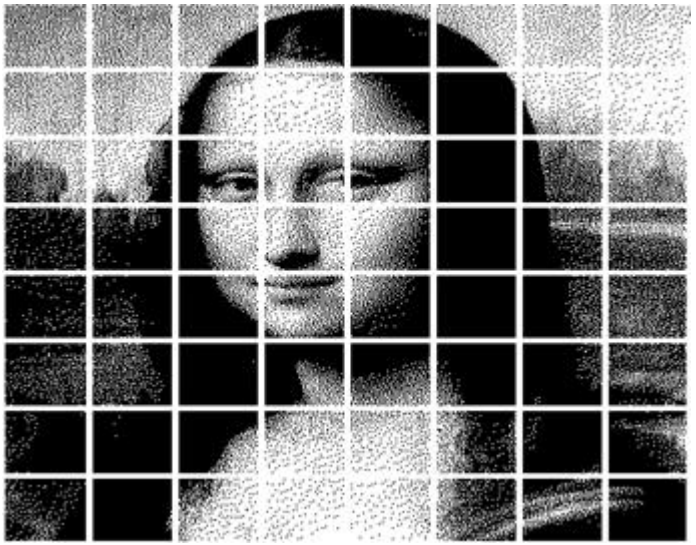
UE1



Task Scheduling

- The *task* is now the update of one sub-domain
- Tasks need to be mapped to UEs
- One task (subdomain) per UE which is the simplest case
- Several subdomains per UE
  - May improve load balance
  - Harder to synchronise
  - A variety of options such as cyclical or linear





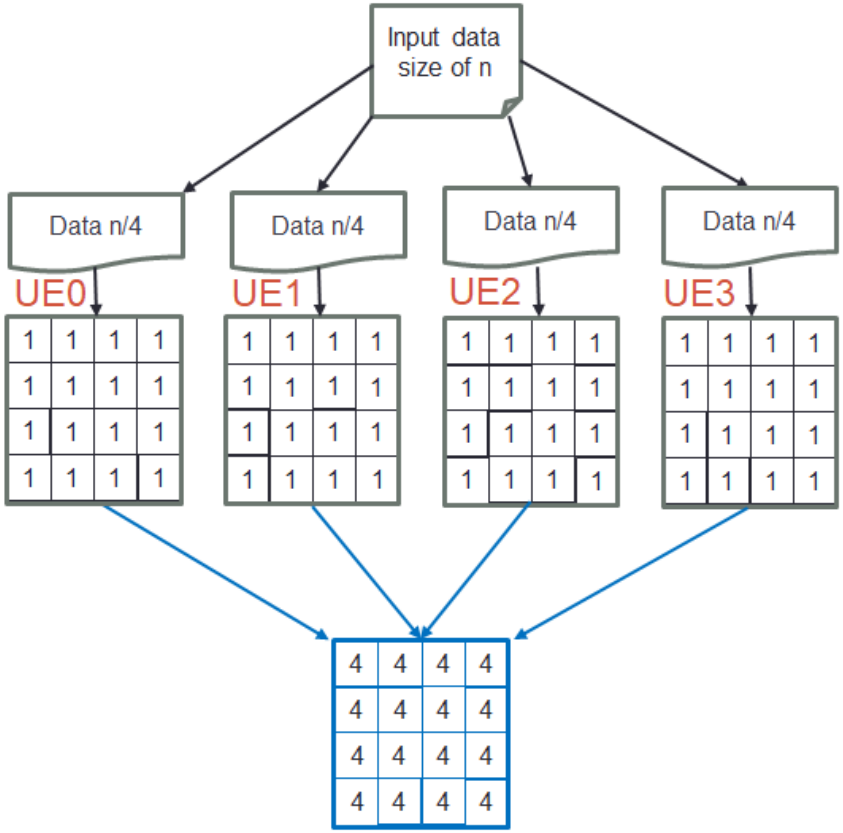
Program Structure

- Geometric Decomposition can be used with either the *Loop Parallelism* or *SPMD supporting structures*
  - In the first practical we used SPMD, we will study the same example using loop parallelism in a later practical.

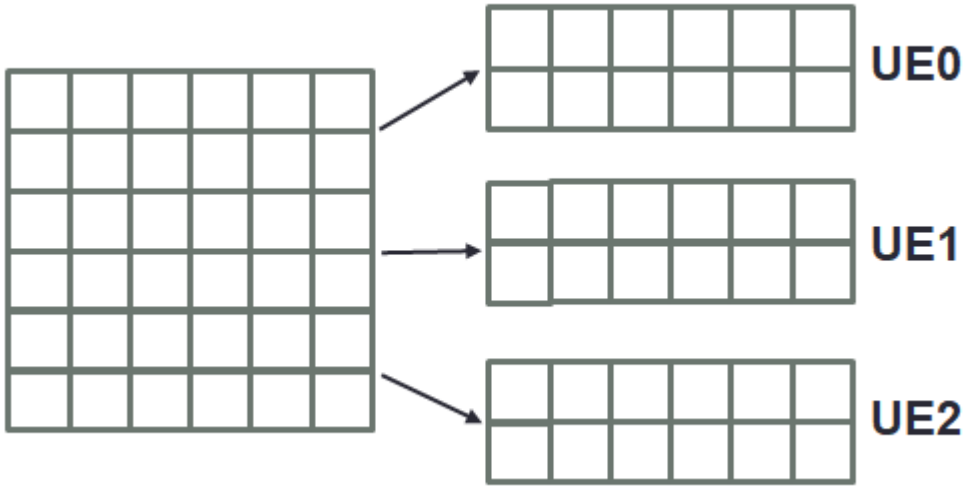
- With *Loop Parallelism*:
  - *An iteration of the loop corresponds to an update of one subdomain in the system*
  - *Maps well onto OpenMP*
- *SPMD*
  - *One process per subdomain*
  - *Exchange operation corresponds to communication between processes*
  - *Maps well onto MPI*

Some examples of geometric decomposition in the real world

BGS Geomagnetic model example



- Input data around 5 million elements of double precision (8 bytes) = 38 MB
- Matrix is n by n double precision (8 bytes)
  - 10000 = 800 MB
  - 20000 = 3.2 GB
  - 50000 = 20 GB
  - 100000 = 80 GB

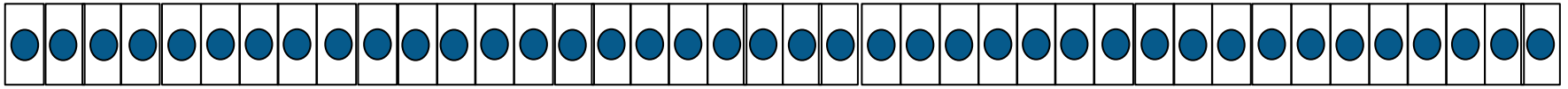


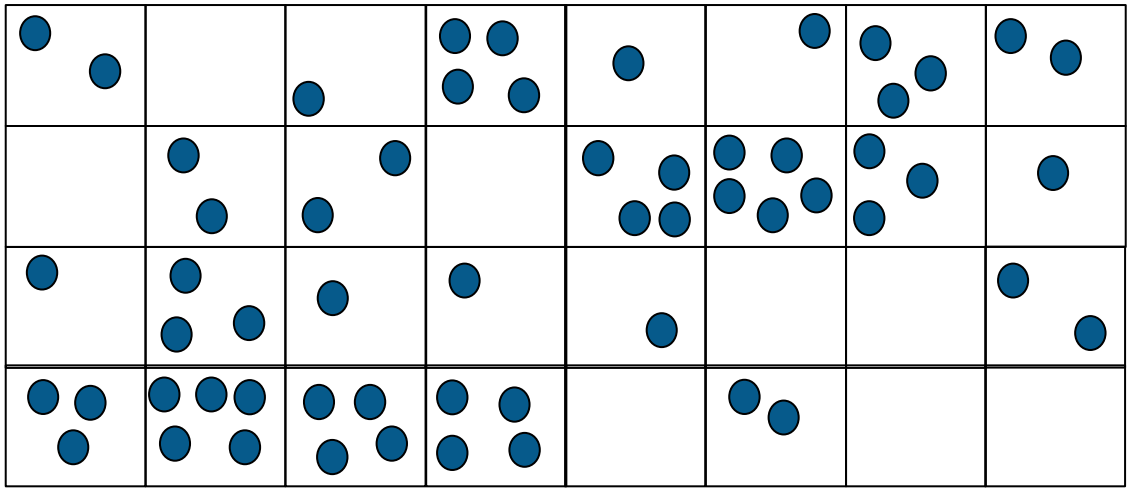
Met Office NAME example

- Numerical Atmospheric Modelling Environment
  - Used for modelling dispersion of particles such as volcanic ash, chemicals and pollution
  - Currently serial and takes a very significant amount of time to run (days)
  - They want to use this model in a far more real time, responsive approach

Met Office NAME example

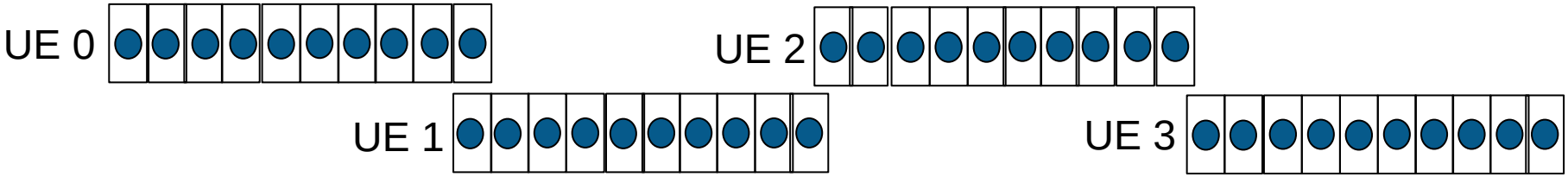
- Three main areas of computation
  - Particle loop, working on properties of each individual particle
  - Particle update loop, updating properties of each particle with some values
  - Chemistry loop, working in grid boxes and calculating chemistry terms based on all the particle's properties in that box & then write back.

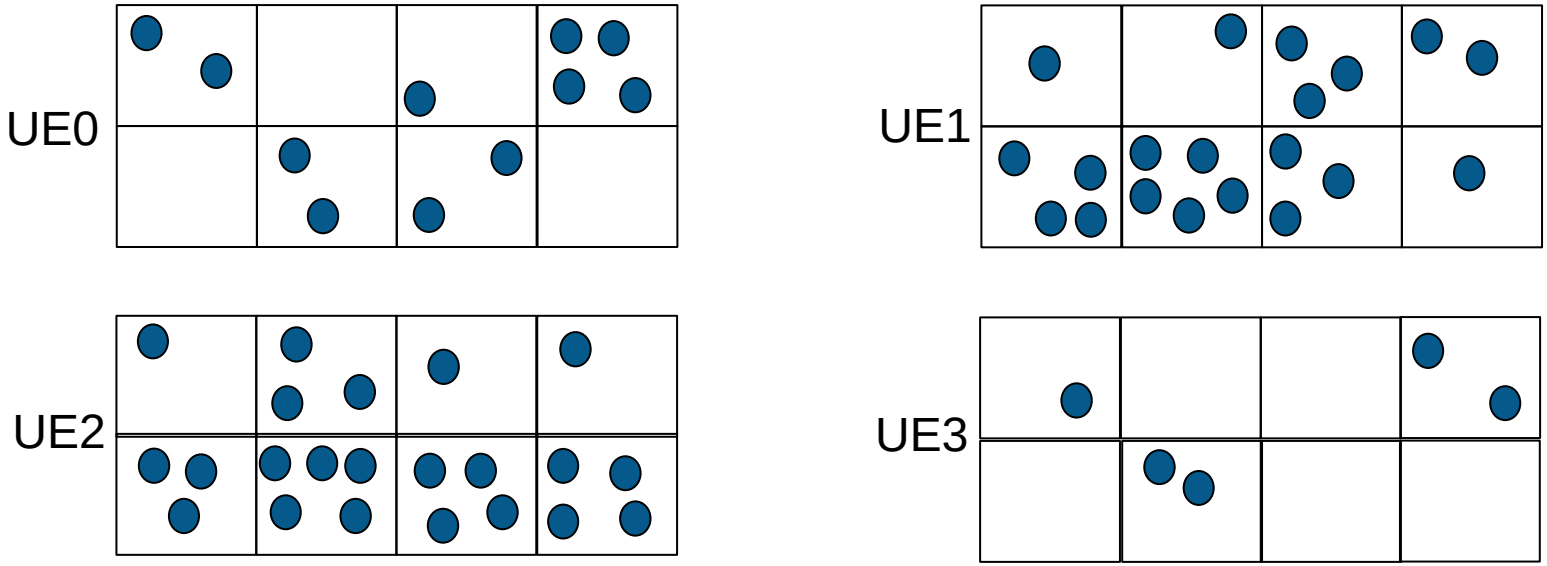




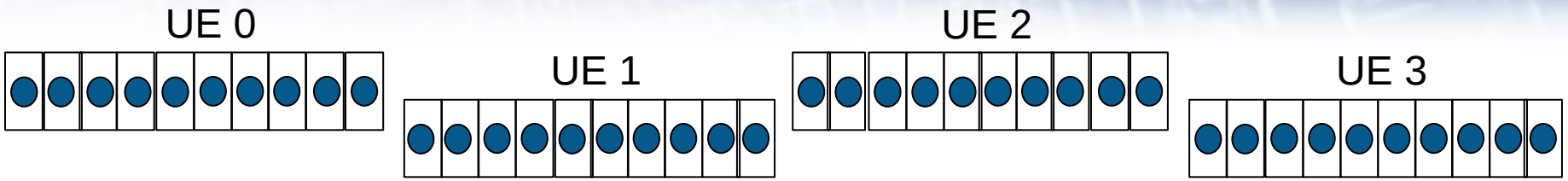
Met Office NAME example

- Three main areas of computation
  - Particle loop, working on properties of each individual particle
  - Particle update loop, updating properties of each particle with some values
  - Chemistry loop, working in grid boxes and calculating chemistry terms based on all the particle's properties in that box & then write back.





Two choices for parallelism

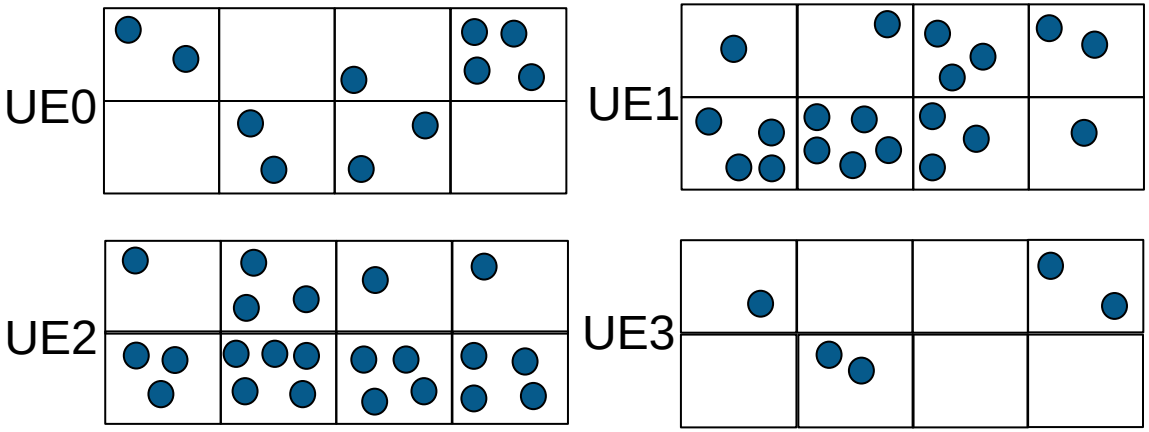


Advantages

1. Simple 1D decomposition of the array
2. An even balance of load in the particle loop & update loop without further work

Disadvantages

1. Communications from every UE needed to determine contributions for particles in each gridbox and write back in chemistry routine



Disadvantages

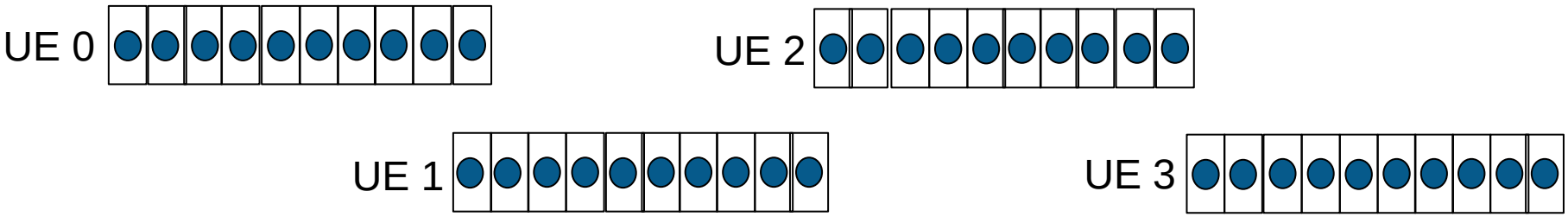
1. There could be a large load imbalance, gridboxes need to be moved between UEs & irregular decomposition
2. Particles move between UEs as they move in the atmosphere

Advantages

1. No communication needed in chemistry to calculate contributions to gridboxes and write results back to the particles as these are local

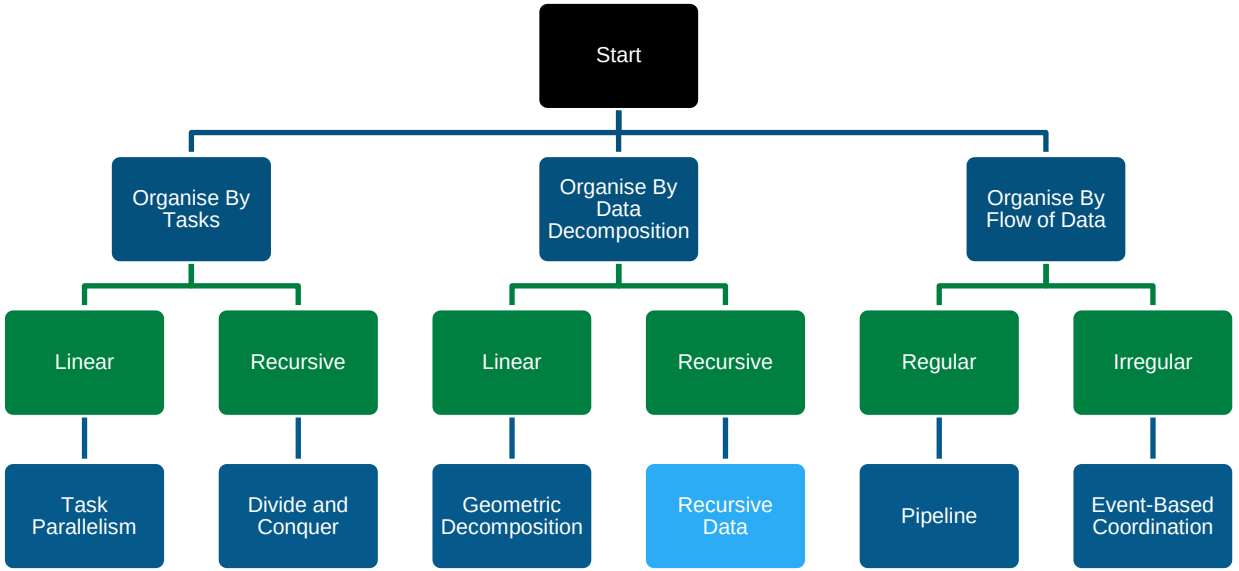
The choice we made

- To go with option 1 and split on the particle array irrespective of geographical location
  - Simpler from a code perspective
  - Get load balancing for free
  - Particle contributions for gridboxes can be encoded as reductions (or even an allreduce) which is implemented efficiently on modern machines
  - With the second approach many P2P communications would be needed, probably with many small messages and complex load balancing undertaken.



Conclusion

- Geometric decomposition is a very common pattern
- A number of choices when implementing this to help tune for performance and scalability
- If you're simulating a physical system where most (if not all) interactions are local then a geometric decomposition is usually the best algorithm strategy
  - Also maps closely on to problems modelled as (discretised) differential equations



Pattern:

RECURSIVE DATA

Recursive Data – Problem

- Given a problem described by an algorithm which involves moving through a data structure in a seemingly sequential way, how can the algorithm be modified to expose parallelism?

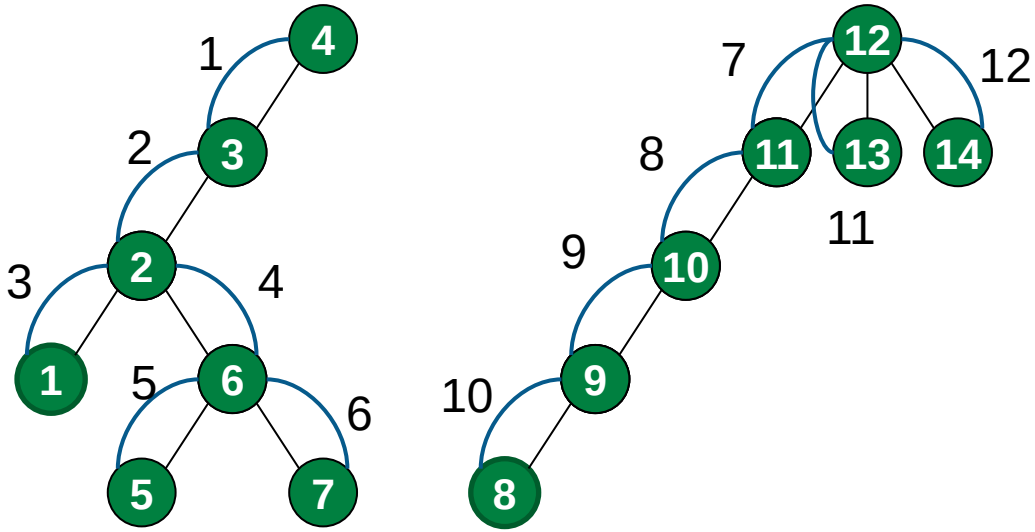
Recursive Data – Context

- Many problems with recursive data structures can be solved with Divide & Conquer
  - If this can be used, use it.
  - Some other algorithms appear to have to move sequentially through the data structure and computing the result at each element.

- It's often possible to re-cast a calculation so that instead of acting on each element in the data structure in turn, the operations are modified so as to expose parallelism
- Also referred to as *Pointer Jumping* or *Recursive Doubling*

Recursive Data – An Example

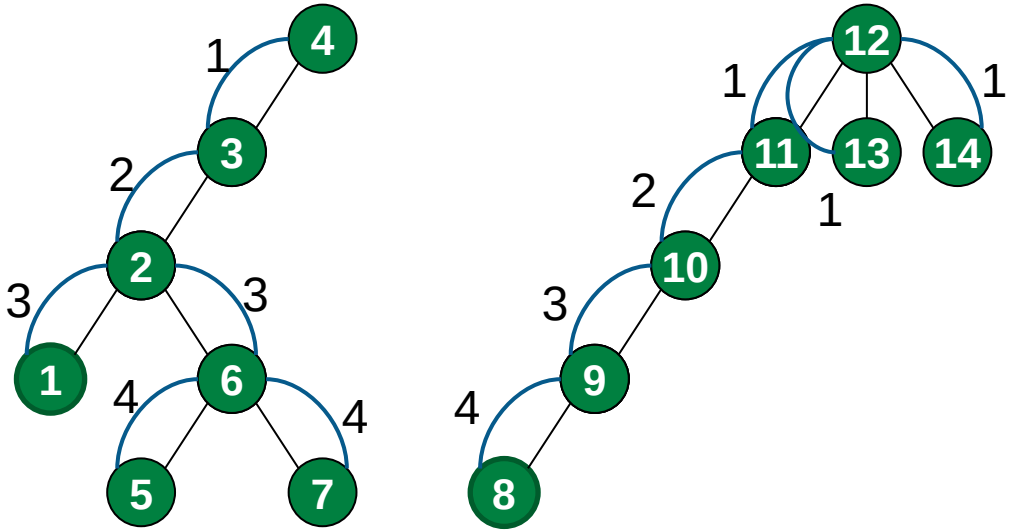
- Finding Roots in a Forest
  - For each node compute the root of the tree containing that node
  - Example from J. Já Já, *An Introduction to Parallel Algorithms*, Addison-Wesley (1992)
- Sequentially (DFS):



- O(N) execution time

Recursive Data – An Example

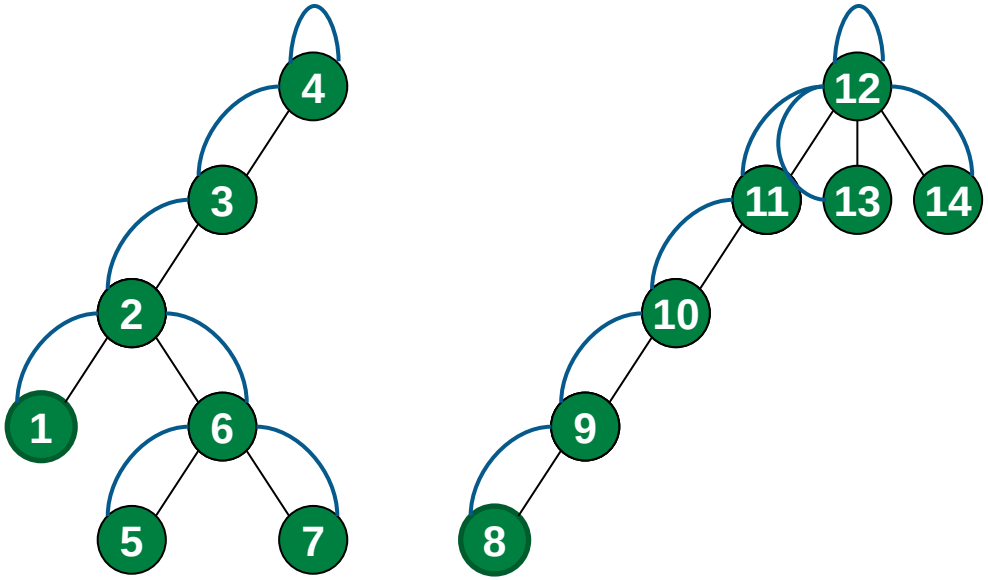
- Naive parallelism where we could operate on subtrees in parallel but can not operate on all element concurrently because how can we find the root of a node without knowing its parent's root



- But heavily reliant on the structure of the tree and still not great

Recursive Data – An Example

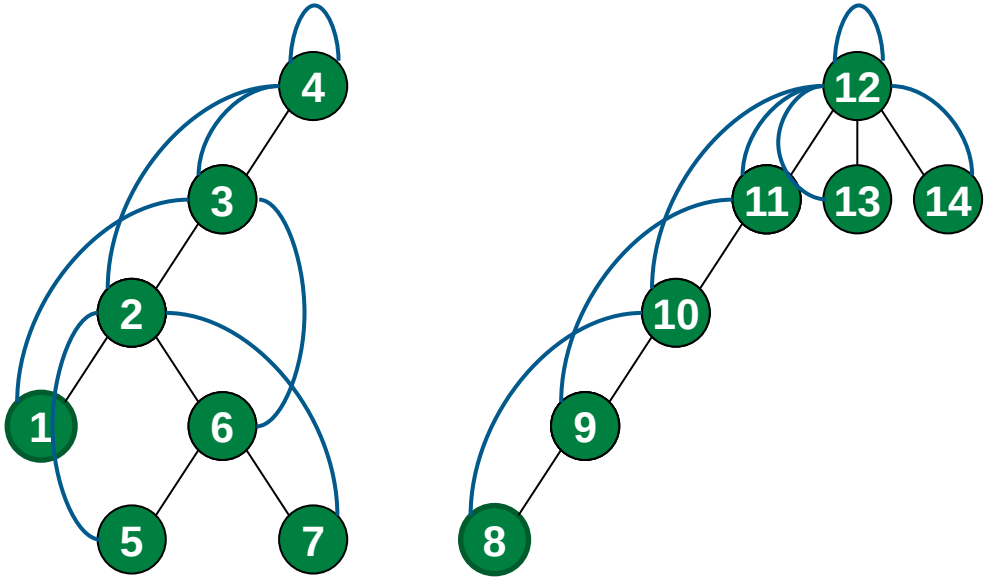
- Let's rethink the problem
- Step 1 – Compute the one hop (direct) parent of each node



- Here each element can be worked on concurrently (we can therefore have 14 tasks)

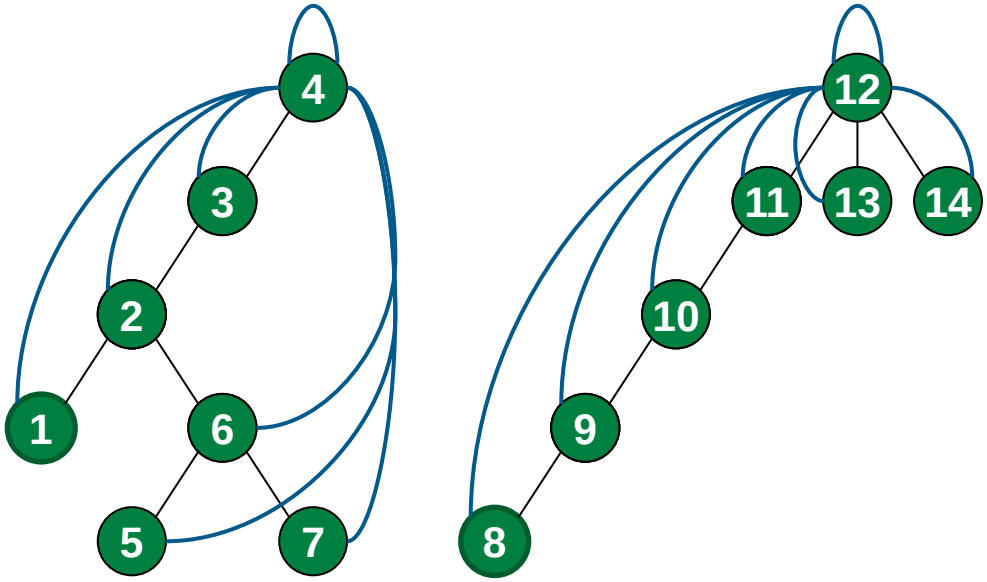
Recursive Data – An Example

- Step 2 – Compute the parent's parent (2 hops away) if applicable



Recursive Data – An Example

- Step 3 – Compute the 3 hops away if applicable



- The algorithm contains much more work than the sequential one O(N log N) vs O(N) but runtime is now O(log N)
- By reshaping the algorithm we have exposed additional concurrency

Recursive Data – Forces

- Recasting the problem to ensure that parts of the data structure can be operated on independently usually increases the total amount of work to be performed
  - This is a trade-off that has to be considered
- Recasting the problem may be difficult
  - In some cases may even be impossible
  - Often results in less intuitive design
    - Can be harder to understand and maintain
- Parallelism exposed may not be efficiently exploitable
  - e.g. the result could be too fine-grained or require excessive communication

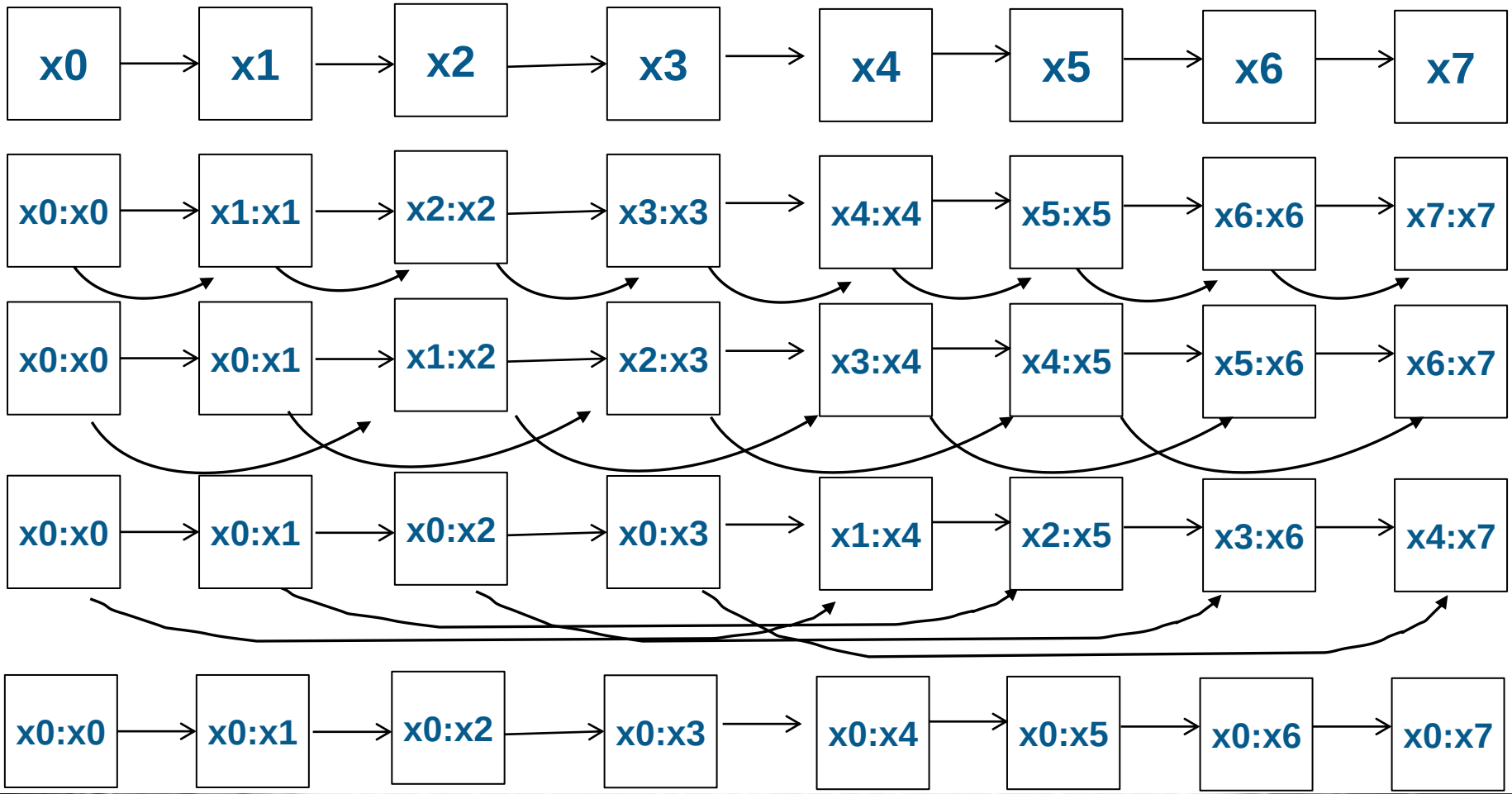
Recursive Data – Solution

- A general solution is difficult to express, but generally consists of
  - Starting from a single element of the data structure
  - Try to determine a means of finding the solution for that element of the data structure by a technique that does not involve waiting for the neighbouring data structure to return a full solution, e.g.,
    - Iteratively follow pointers of neighbouring elements without actually waiting for them to have computed their ultimate result
    - Build up a final result from smaller calculations that can be performed locally
- Features of the solution
  - Data decomposition: Usually one element of data structure per UE
  - Structure: Typically a loop of iterations; operate simultaneously on every element once each iteration. Typical operations include “replace each element’s successor with its successor's successor.”
  - Synchronisation: Typically at end of each iteration (manual or implied)

Example: Partial sums of a linked list

```
k=pid()
temp[k]=next[k]
```

```
while temp[k] != null {
  x[temp[k]]=x[k]+x[temp[k]]
  temp[k]=temp[temp[k]]
}
```



A word of warning with this pattern…..

- As the work required goes from O(N) to O(N log N) we can get caught out by this if we don't have enough UEs
  - i.e. N=1024, time per step is t. Therefore sequentially it would take 1024 * t .
  - Total work with this pattern is O(N log N) = 1024 * 10 * t = 10240*t
  - With 1024 UEs, the total runtime is 10*t
  - But, if we only have 2 UEs, then the runtime is 5120*t
  - In this example the break even point is 10 UEs, therefore carefully consider if the pattern is worth applying

- Potential best scaling can sometimes be limited, but often preferable to running in serial

Summary

- Organising parallelism based upon the decomposition of data is very common
  - Especially the geometric decomposition pattern
- Has formed a critical cornerstone of HPC codes for many years
  - Certainly will continue to be a very important approach into the future
- However there are challenges for running these codes on millions of cores
  - In terms of scalability and resilience
  - The research space is currently looking for alternatives (specifically organising parallelism around tasks) which can help with this
  - More about this in the next lecture!