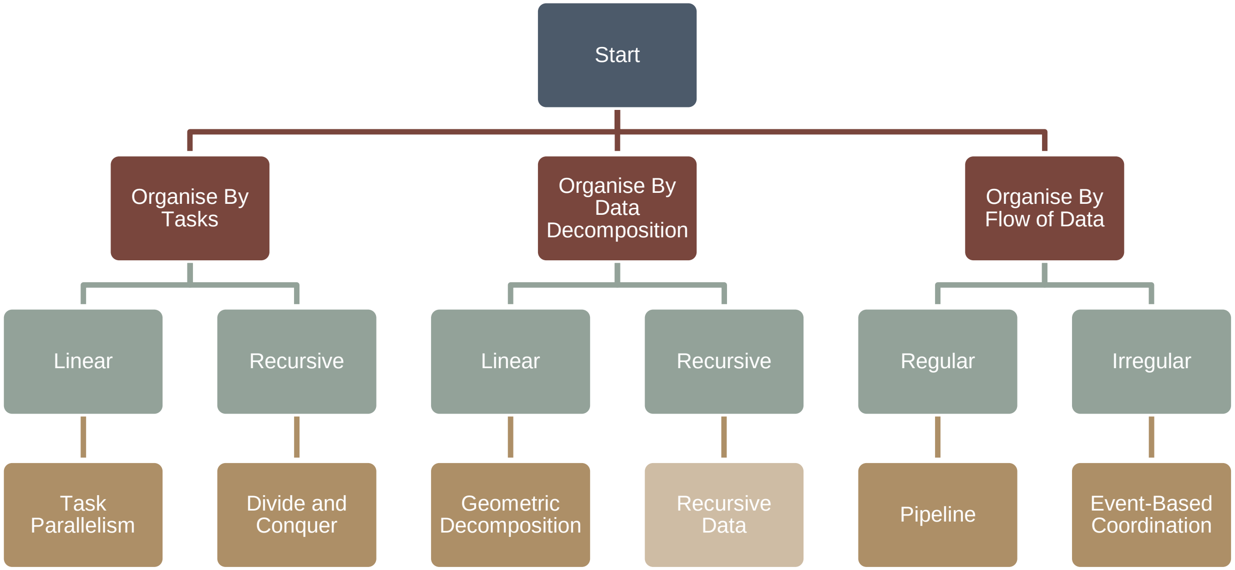
Parallel design patterns ARCHER course

Recursive data, task parallelism,
divide and conquer

Reusing this material



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?

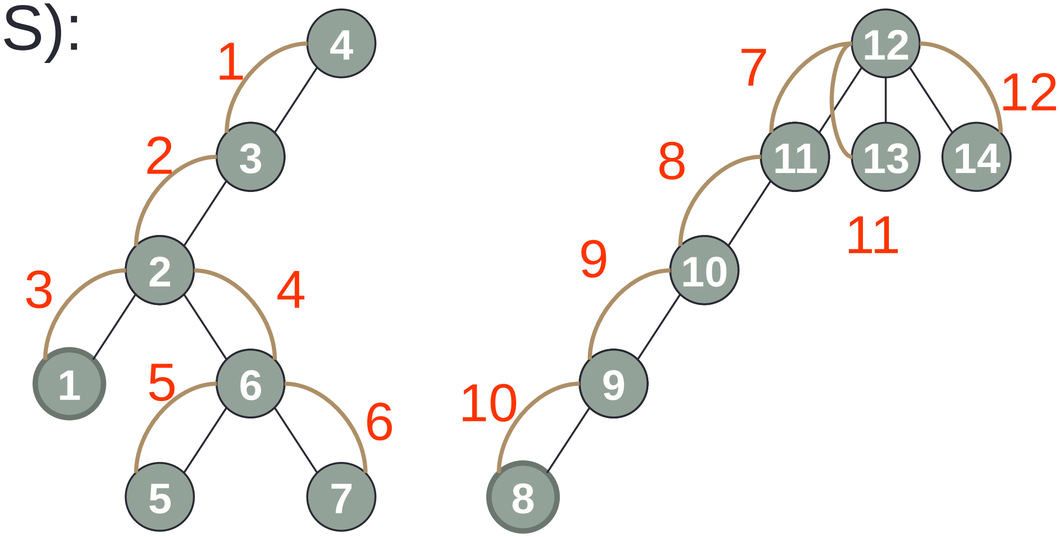
Recursive Data – Context

- 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

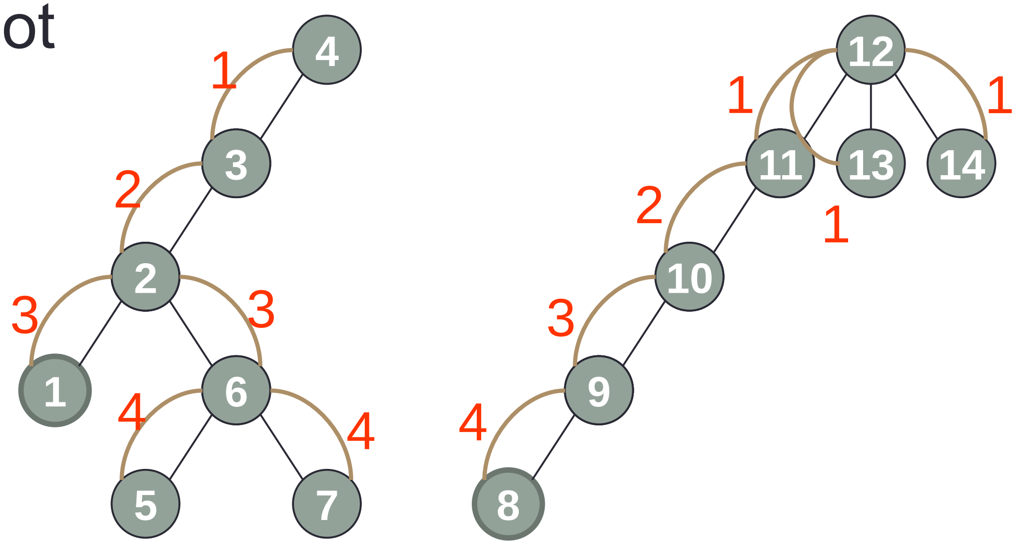
- Finding Roots in a Forest
  - For each node compute the root of the tree containing that node
  - Example from J. Já Já, *An Introduction to Parallel Algorithms*, Addison-Wesley (1992)
- Sequentially (DFS):



- O(N) execution time

Recursive Data – An Example

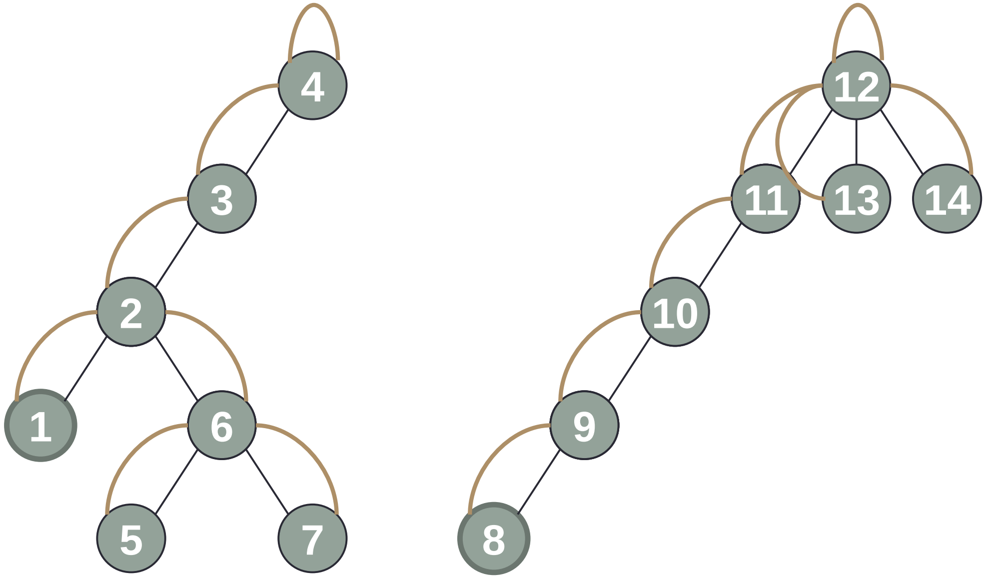
- Naive parallelism where we could operate on subtrees in parallel but can not operate on all element concurrently because how can we find the root of a node without knowing its parent's root



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

Recursive Data – An Example

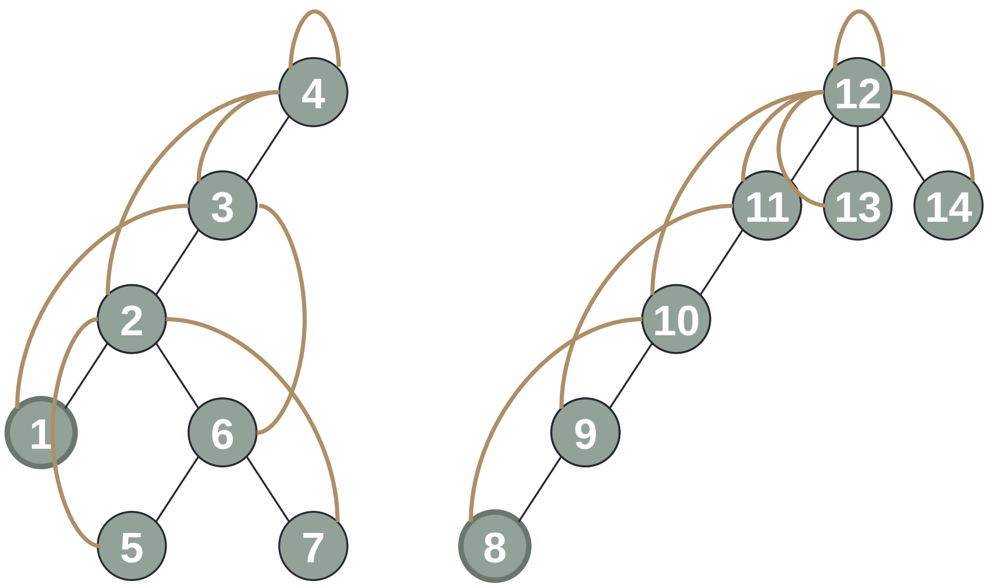
- Let's rethink the problem
- Step 1 – Compute the one hop (direct) parent of each node



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

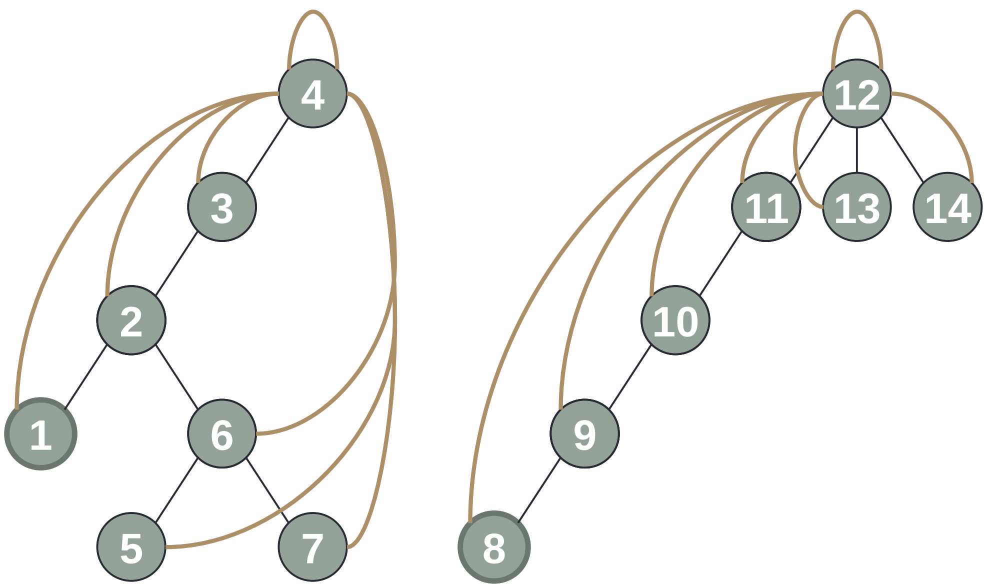
Recursive Data – An Example

- 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

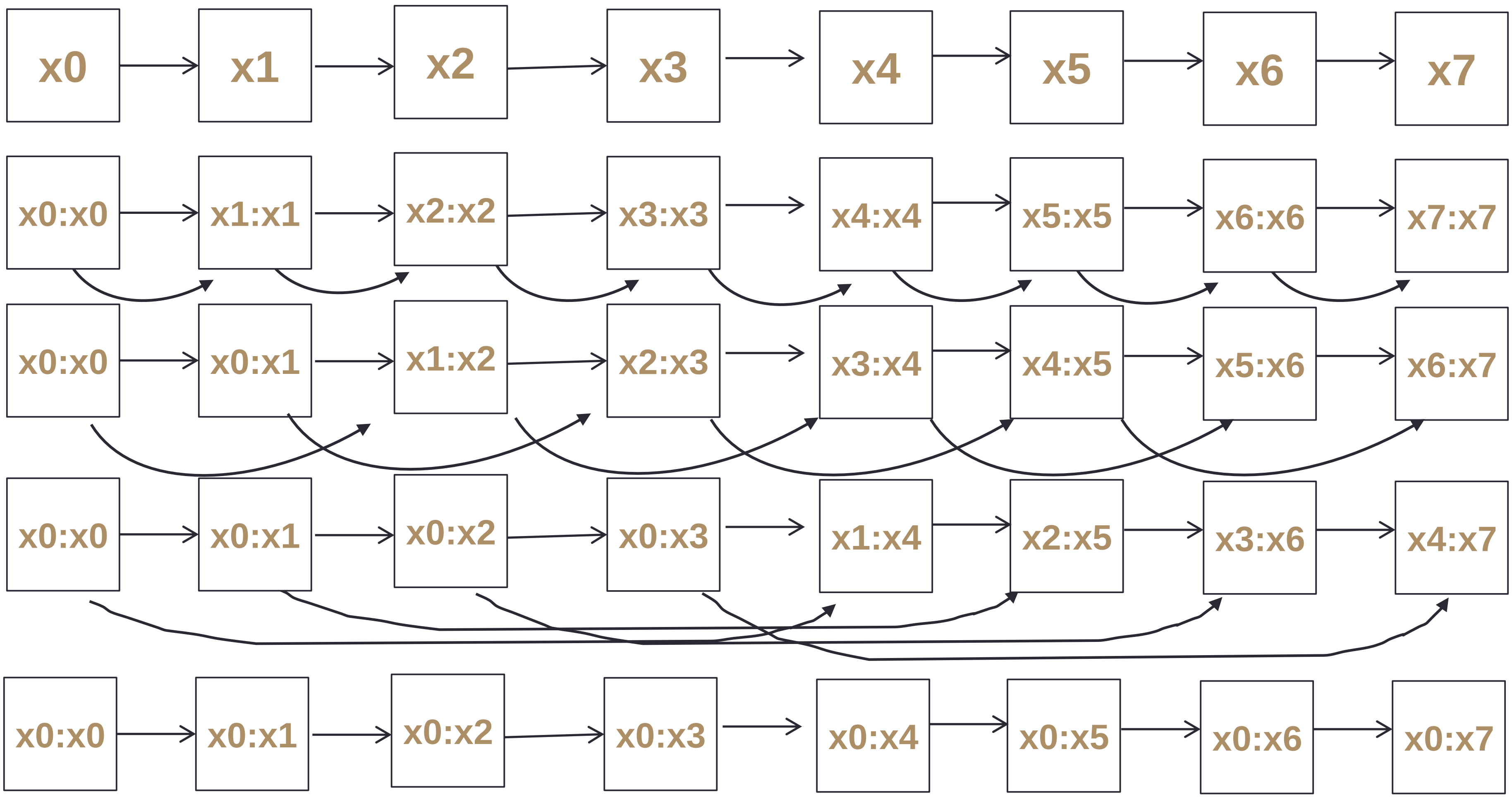
- Recasting the problem to ensure that parts of the data structure can be operated on independently usually increases the total amount of work to be performed
  - This is a trade-off that has to be considered

- 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



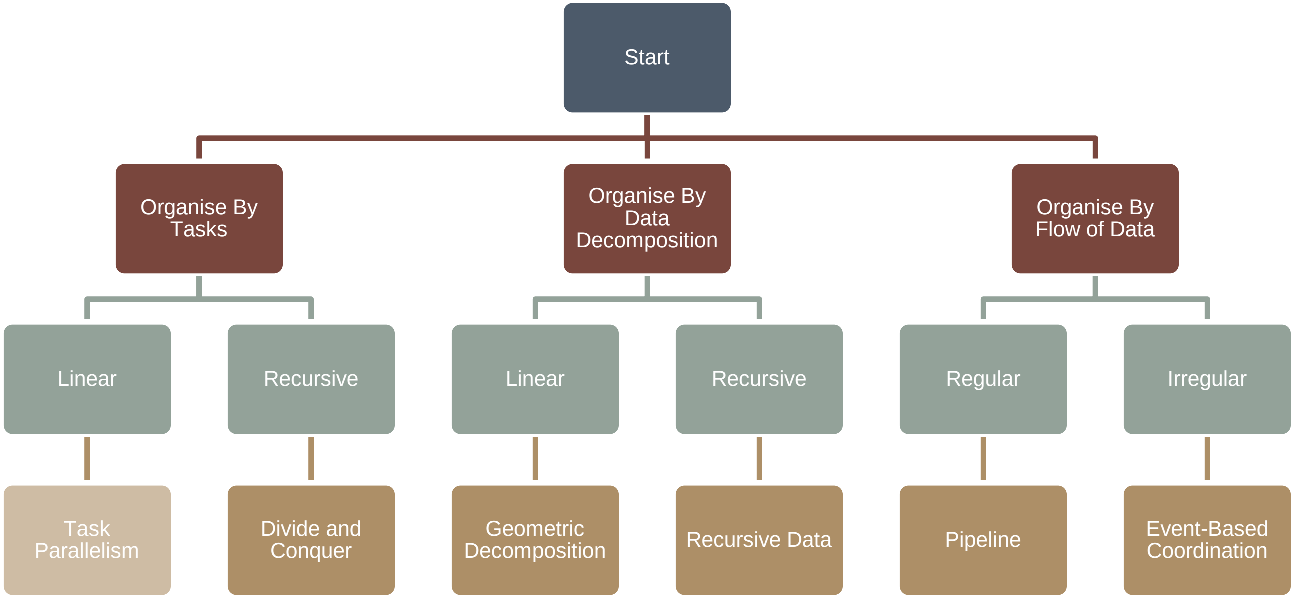

```
k=pid()
temp[k]=next[k]
```

```
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



TASK PARALLELISM

“Task Parallelism”

- Here we focus on the Task Parallelism *Pattern*
- We’re looking at a particular *Problem* in a particular *Context* and its *Solution*
- The phrase is also used in other contexts (with varying but related meanings)
  - A common differentiation is between *“Task Parallelism”* and *“Data Parallelism”*
    - *a more general definition than encompassed by this pattern*

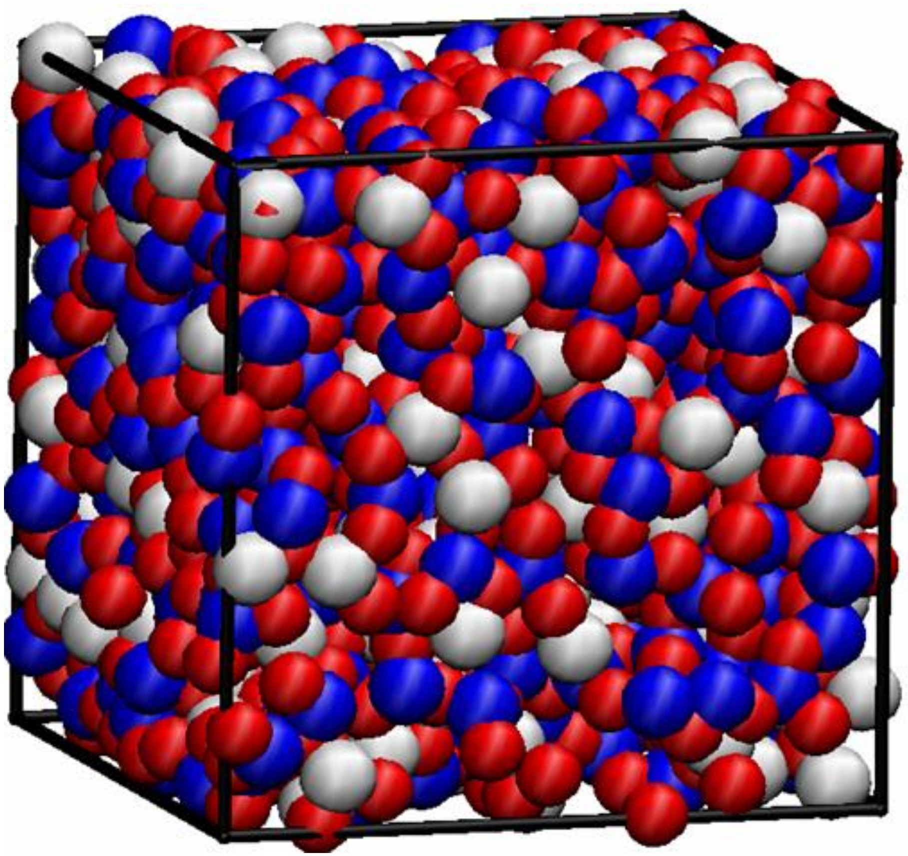
Task Parallelism – Problem

- When a problem is naturally decomposed into a collection of tasks that can execute concurrently, how can this concurrency be exploited efficiently?

Task Parallelism - Context

- All parallel algorithms can ultimately be broken down into concurrent tasks
  - There can be more than one way to do this
- This pattern is about problems that are best dealt with by an algorithm that is *focussed on these tasks and their interactions.*
  - The design is based directly on the tasks
- Arguably this pattern is defined best by what it does not include, namely:
  - Geometric Decomposition (organised by data), Pipeline (organised by the flow of data)
- Tasks can be completely independent, or there can be interdependencies

Examples



- Molecular Dynamics Simulation
  - Often actually uses more than one pattern, but conceptually
    - Moving *n* particles: O(*n*) tasks
    - Calculating the forces between particles: O(n^2) tasks
- Computer game
  - User control
  - Game physics
  - Render
  - AI
  - Music
  - Sound effects

Task Parallelism - Forces

- The same aspects of the problem that influence the pattern to consider are also relevant to how concurrency can be best exploited:
  - Efficiency
  - Simplicity
  - Portability
  - Scalability

- An important consideration here is **load balance**
- Correct management of interdependencies

Task Parallelism – Solution

- Consider each of the following in turn and then together:
  1. Tasks
  2. Dependencies
  3. Schedule
     - How tasks are assigned to processes, threads
       - *Processes & threads referred to as Units of Execution (UEs)*
     - Note that this is still one step away from how these are run on hardware
       - *Hardware elements referred to as Processing Elements (PEs)*

Tasks

- There should be at least as many tasks as UEs
  - Preferably many more
    - Allows more flexibility in scheduling and potentially better load balance

- The computation associated with each task must be large enough to offset overheads like task management and dependencies between tasks

- If your design does not meet these criteria, then can you split in a way that results in more, computation rich, tasks?

Dependencies

- Ordering constrains
  - Task groups must execute in a specific order i.e. we must set the boundary conditions & initial values before computing the initial residual.
  - Could think of the problem as a sequential composition of task parallel groups i.e.

```
(boundary conditions and initial values) ; initial residual ;
          (solution residual and jacobi iteration)
```

- Shared data dependencies
  - Data shared between tasks, ranging from none (embarrassingly parallel) to lots (tightly coupled.)
  - Our practical example isn't too bad, but you do need to exchange neighbouring data

Categorising dependencies

- Removable dependencies
  - Can remove by code transformation
  - E.g. transforming iterative expressions to closed form

```
int ii=0;jj=0;
for (int i=0;i<N;i++) {
  ii++;
  d[ii]=time_consuming_work(ii);
  jj=jj+i;
  a[jj]=large_calculation(jj);
}
```

- ii and jj create a dependency between tasks
- But ii = i
- And jj is the sum of 0 through i

```
for (int i=0;i<N;i++) {
  d[i]=time_consuming_work(i);
  a[(i*i+i)/2]=large_calculation((i*i+i)/2);
}
```

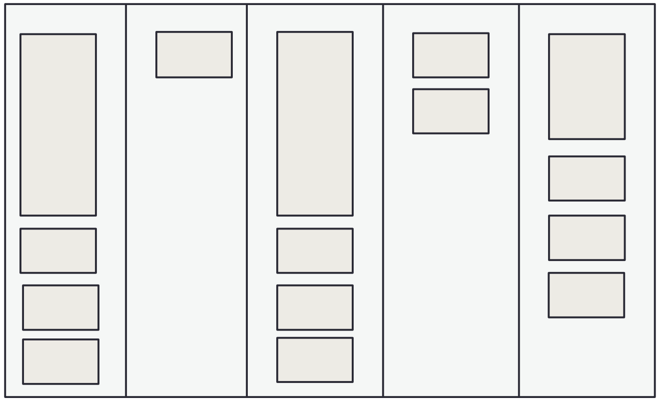
Categorising dependencies

- Separable dependencies
  - When dependencies involve accumulation into a shared data structure
  - Replicate some data at the start of a task: **replicated data**
  - execute task
  - recombine replicated data
    - often a reduction operation
      - reductions supported directly in, e.g., MPI, OpenMP

- Other dependencies
  - If shared data can not be pulled out of the tasks and is read/write then it is difficult
  - Apply Shared Data pattern

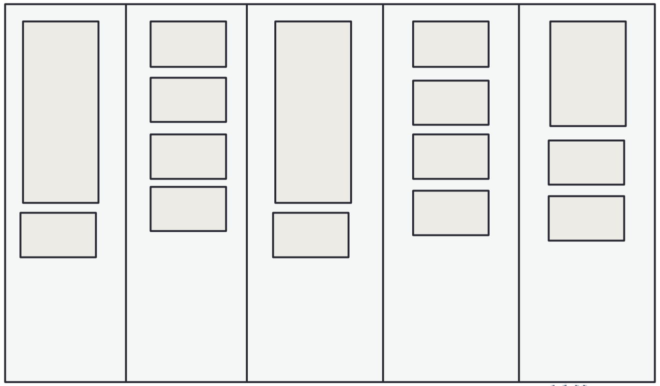
Scheduling

- Closely related to the Implementation Strategy
- Scheduling is critical to load balancing
  - Schedules can be *static* or *dynamic*



Poor load balancing

- Static scheduling
  - useful for regular, predictable workloads
  - can also be useful for more “random” loads by using round-robin allocation
- Dynamic scheduling can be done with, e.g. task queues, work stealing
  - Helpful when not all tasks are known in advance

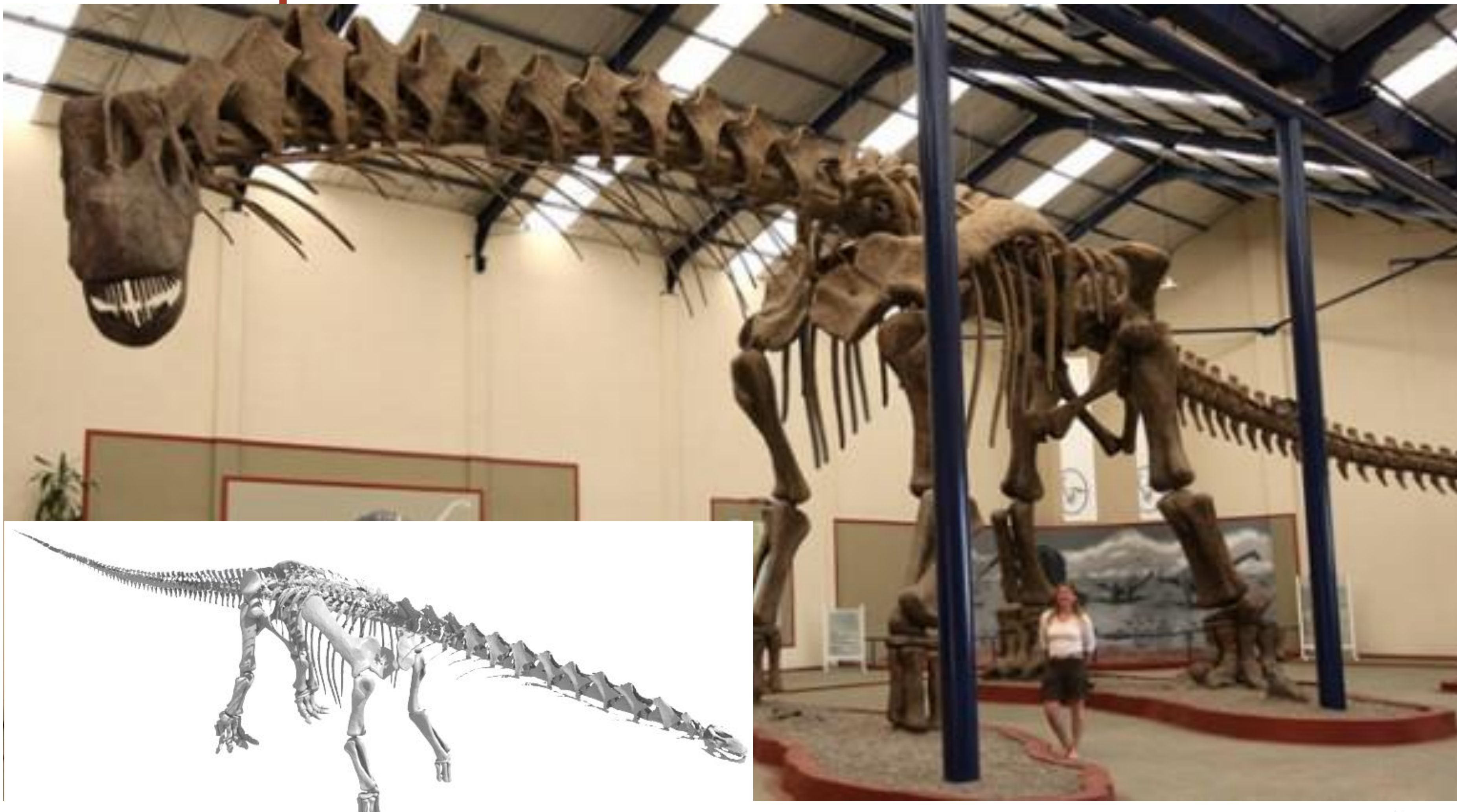


Better load balancing

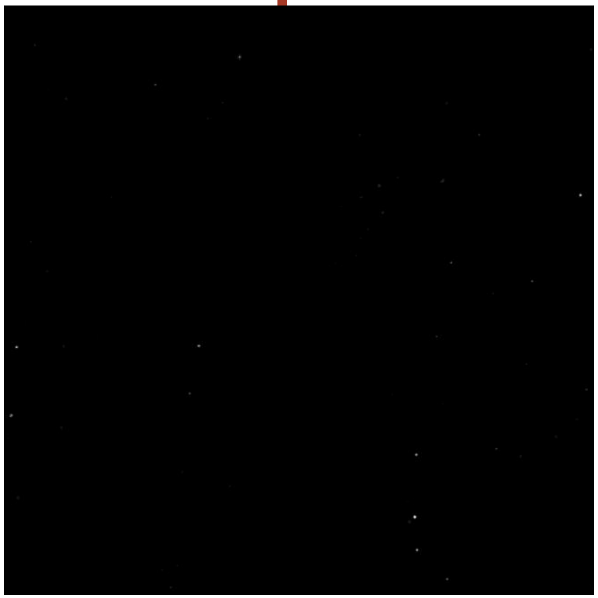
Task Parallelism: Languages & Architectures

- Task Parallelism can be done with nearly all parallel languages
  - The decision between, say, OpenMP and MPI is more likely to be based on the chosen Implementation Strategy
- Explicitly data-parallel languages such as HPF are an exception, although (contrived) solutions exist to use HPF
  - External libraries
  - Mixed-mode with MPI
- Often map well onto loop parallelism, master/worker or SPMD implementation strategies.

Example: Dinosaurs

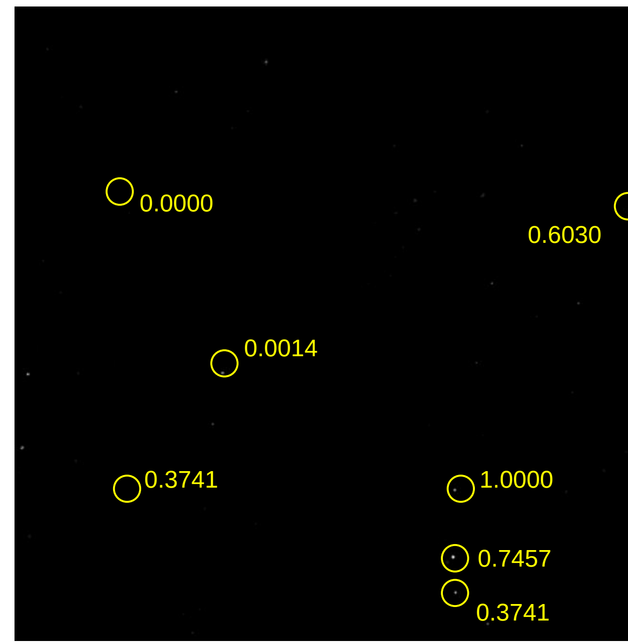


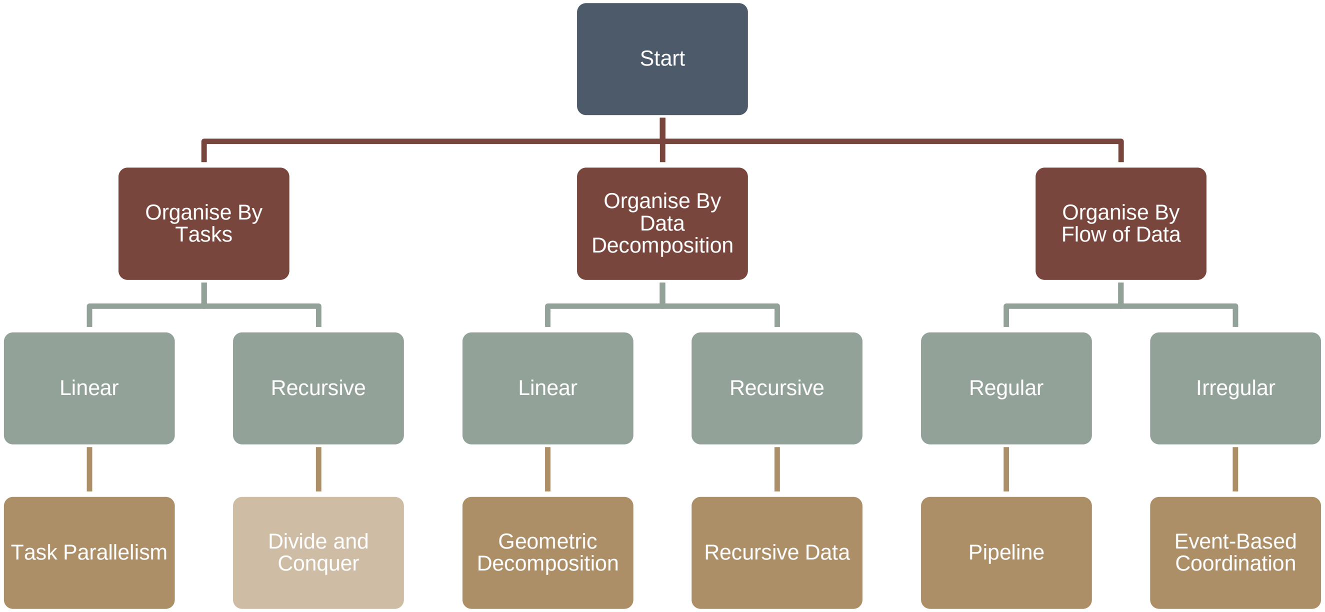
Example: Star Extractor



- Each input image is run as a concurrent, independent task
  - Identifying objects and classifier neural network
- The classifier neural network can operate on each object as an independent task

1	134.0376	292.1414	......	0.0000
2	239.6541	192.4977	......	0.0014
3	307.1008	305.6235	......	0.5181
4	319.4861	263.6567	......	1.0000
5	263.3937	58.7983	......	0.7457
6	171.7773	120.8677	......	0.3741
7	16.1523	31.4022	......	0.6030





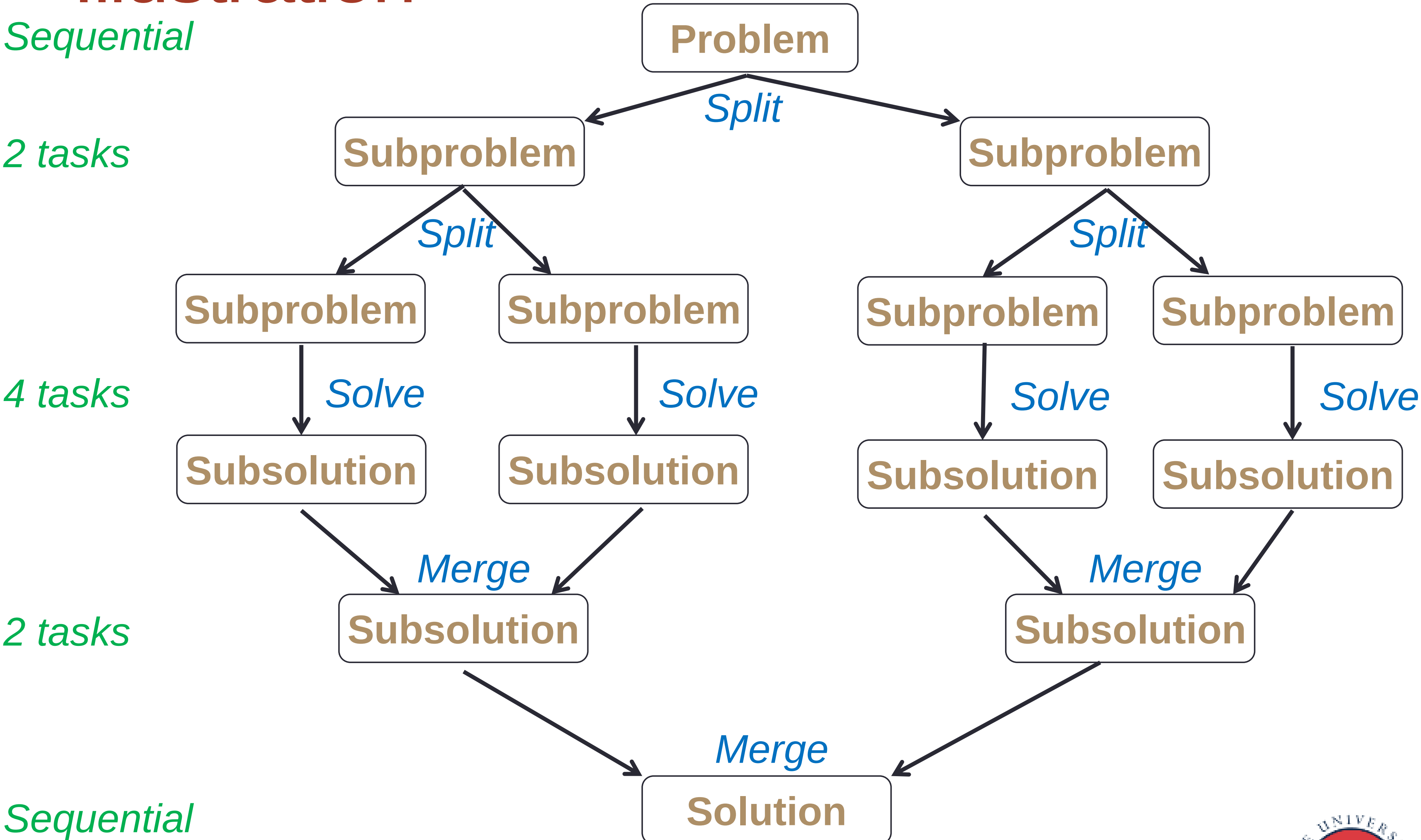
DIVIDE & CONQUER

Divide & Conquer - Problem

- Given a problem which can be solved by solving sub-problems and combining their results together, how can this concurrency be exploited by a parallel algorithm?

- Divide & Conquer is sometimes referred to as *recursive splitting*
  - but note that this is different from the Recursive Data pattern

Illustration



Divide & Conquer - Context

- Divide-and-conquer is used in many sequential algorithms
- Basic strategy:
  - Split problem into smaller sub-problems
  - Solve smaller sub-problems
    - These sub-problems can often, in turn, be split.
  - Merge solutions
- Parallelism comes from observation that sub-problems are typically independent and can be solved concurrently
- Many problems expressed mathematically map well into divide and conquer approaches

Divide & Conquer - Forces

- Obvious exploitable concurrency, but not always easy to exploit *efficiently*

- Exploitable concurrency often varies throughout lifetime of program (especially with recursion)
- Amdahl's law states that the serial fraction constrains the speed up – *therefore the split and merge should be trivial*.
- Problems are typically "created" and "solved" on different UEs resulting in need for communication, and often movement of data – if the number of tasks are too large then can the cost of parallelism swamp speed up?

Divide & Conquer – Solution

- In serial, divide & conquer often implies recursive calls:

```
begin solve(problem)
  if problem small enough
      return solveBaseCase(problem)
  else
      split(problem, subproblem1, subproblem2)
      solution1=solve(subproblem1)
      solution2=solve(subproblem2)
      return merge(solution1,solution2)
end solve
```

- Parallelise by making each call to `solve` a task

Divide & Conquer: Other considerations

- In serial, the base case is usually the smallest possible subdivision and trivial to solve (e.g. sort one number)
- In parallel, size of the smallest subdivision should be chosen for performance (and should be tuneable). Consider:
  - communication / transfer of data between task and sub-task
  - size of problem: e.g. stop splitting when subproblem fits in cache
- If subtask is on a separate PE then it might make sense to duplicate some shared data
- If tasks are not independent, also use *Shared Data* pattern
- It might make sense to split into more than two subtasks
  - e.g., if it's easier to do one big merge than two smaller merges (which can in turn depend on whether a merge can be parallelised)

Divide & Conquer – Implementation

- Take the tasks and solve these using
  - Fork/Join pattern (see lecture and practical tomorrow), or
  - Master/Worker pattern (see lecture and practical tomorrow)

- **Fork/Join** works well with regular problems
  - One task splits the task in two and forks off a subtask (or subtasks) to solve the problem, it waits for the subtasks to complete, then joins with the subtasks to merge the solution
- **Master/Worker** works well with irregular problems
  - Maintain a queue of tasks and a pool of UEs which take tasks from the pool when they become free
  - Slightly more complex but often gives better load balance if the tasks have unpredictable work loads

Example: Mergesort

- Well known sorting algorithm based on divide and conquer.
  - There is a certain threshold, smaller than this then sort the array sequentially (i.e. using some algorithm such as quicksort)
  - In the split phase the array is split by partitioning it into two subarrays of size N/2
  - Apply mergesort procedure recursively to sort subarrays
  - In merge phase the two (sorted) subarrays are combined

- The algorithm lends itself to parallelisation by doing the two recursive mergesorts in parallel

Example: Mergesort

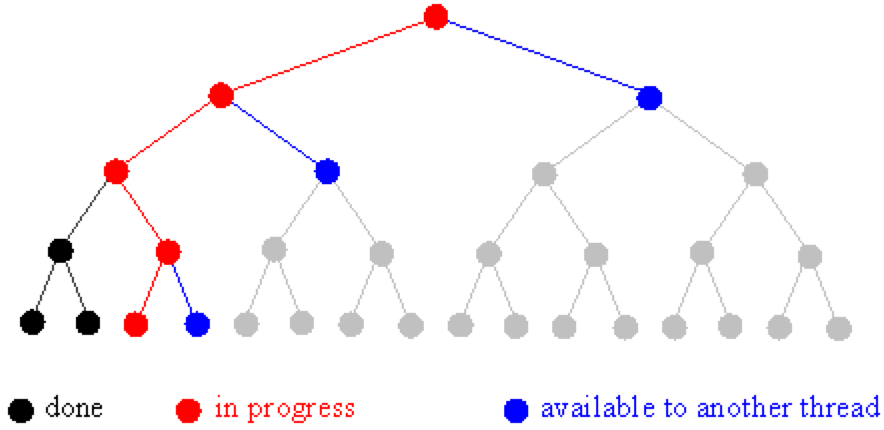
```
sort(int[] A) {
  if (length(A) < THRESHOLD) {
    return quicksort(A)
  } else {
    pivot=length(A)/2;
    t=create new task {
      B=sort(A(1:pivot))
    }
    C=sort(A(pivot:length(A))
    wait for t to complete

    return merge(B,C)
}
```

- The merge function is the same as a sequential mergesort.
- The sketch of the algorithm is very similar to the sequential version.

- Carefully consider the efficiency of merge and splitting of the array.
- This is the subject of a later practical

Recursive task parallelism

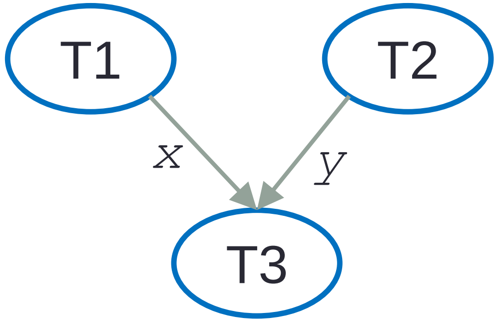
- This was called divide and conquer to represent the general algorithmic pattern
- Recursive task parallelism would probably be a better name nowadays
  - As tasks spawning sub-tasks which themselves spawn sub-tasks etc can be used in a variety of different algorithms
  - These algorithms include divide and conquer, but the same ideas we have discussed can potentially be applied to other algorithms too



Tasks in OpenMP

```
#pragma omp task
{
    ……
}

#pragma omp task
{
    #pragma omp task
    {
      ……
    }
    ……
}

#pragma omp task
{
    ……
}
#pragma omp taskwait
```

```
int x,y
#pragma omp task depend (out:x)
{
    ……
}

#pragma omp task depend (out:y)
{
    ……
}

#pragma omp task depend (in:x,y)
{
    ……
}
```



Conclusions

- Recursive data is pretty uncommon, but might be useful in some situations
- Task parallelism where the tasks are linear and created sequentially
- Divide and conquer when the tasks are created recursively
  - This is when things start to get a bit more complex, because we are in a situation where the number of tasks is non-deterministic, unstructured and unpredictable
  - But the interaction between tasks is predictable (i.e. parent-child)