Parallel Design Patterns-L04

Task Parallelism,

Divide & Conquer

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

Finding Concurrency

- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

Algorithm Structure

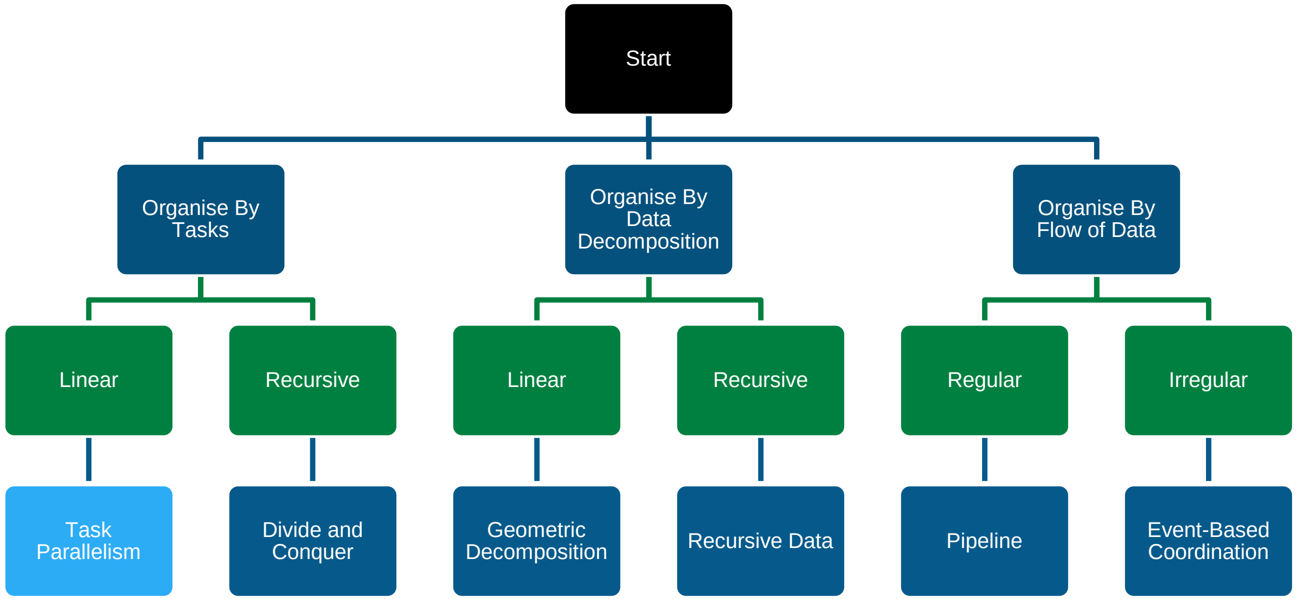
- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, …

Supporting Structures

- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

Implementation Mechanisms

- UE Management, Synchronisation, Communication, …



Pattern:

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

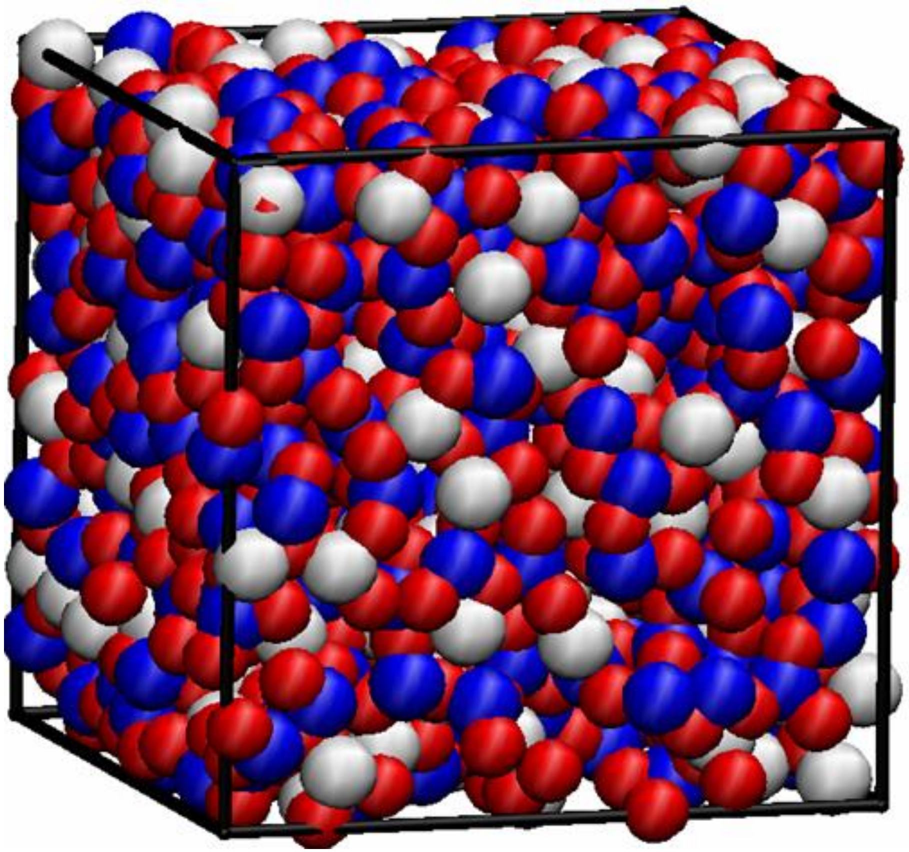
- When a problem is naturally decomposed into a collection of tasks that can execute concurrently, how can this concurrency be exploited efficiently?

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 sometimes referred to as Units of Execution (UEs)
     - Note that this is still one step away from how these are run on hardware
       - Hardware elements sometimes 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?

Task types

- Closures
  - Task contains a specific piece of functionality. It will execute, possibly update some values and then finish.
  - Typically wouldn’t spawn sub-tasks
  - No communication needed inside a task

- Continuation
  - A long running task, often for the entire run of the code
  - Might spawn many sub tasks and wait for these to complete
  - Communication might be issued inside the task

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 || initial values) ; initial residual ;
          (solution residual || 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);
}
```

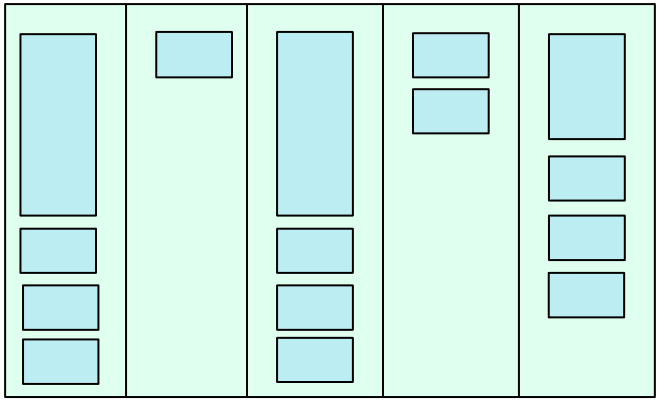
Categorising dependencies

- 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

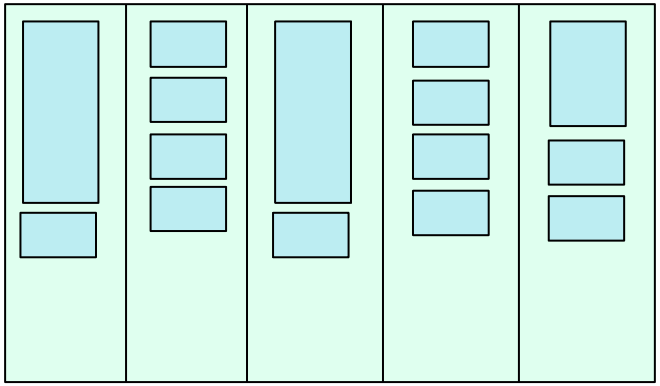
Scheduling

- 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



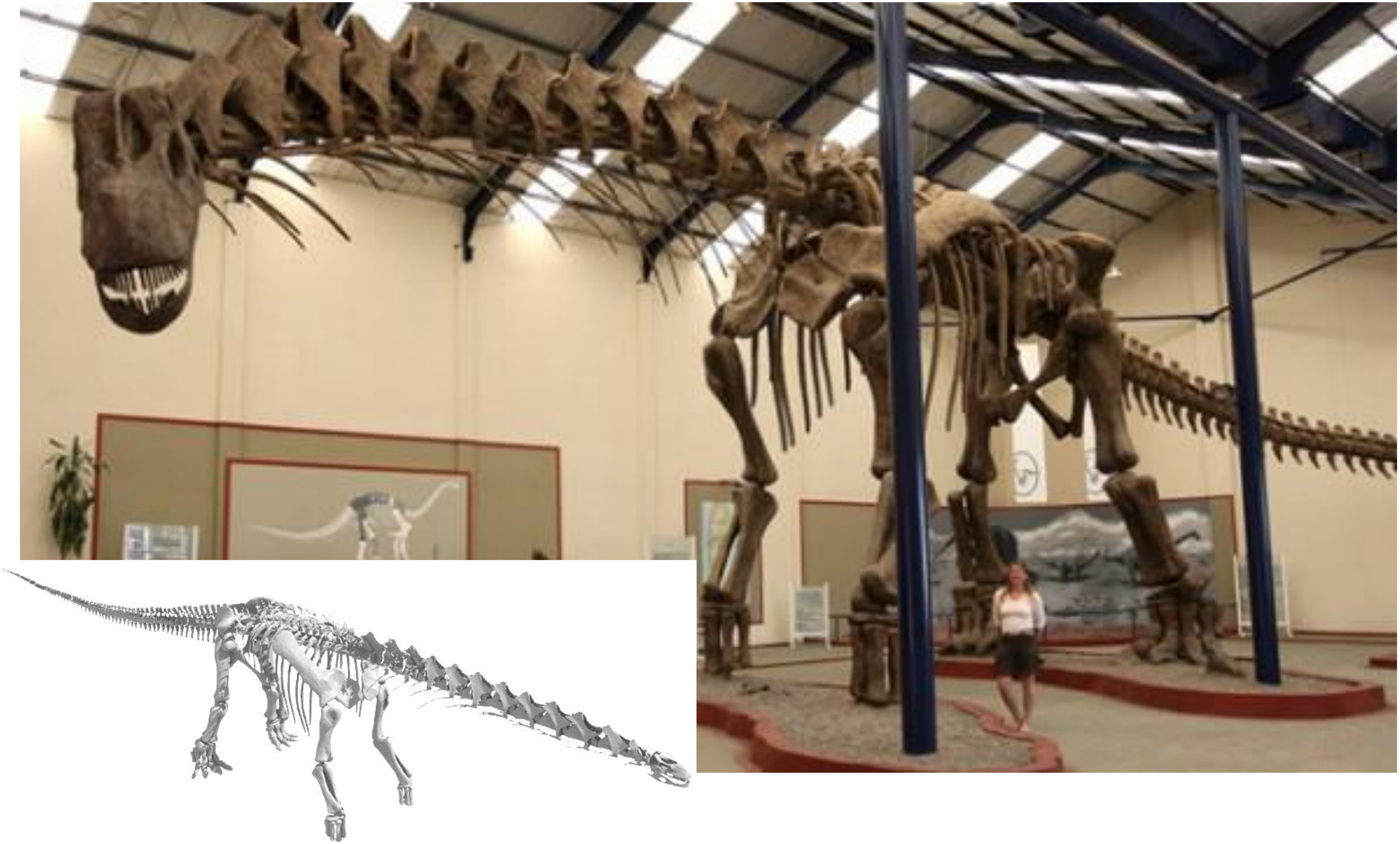
Better load balancing

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 (lecture 10), master/worker (lecture 9) or SPMD (lecture 9) implementation strategies.

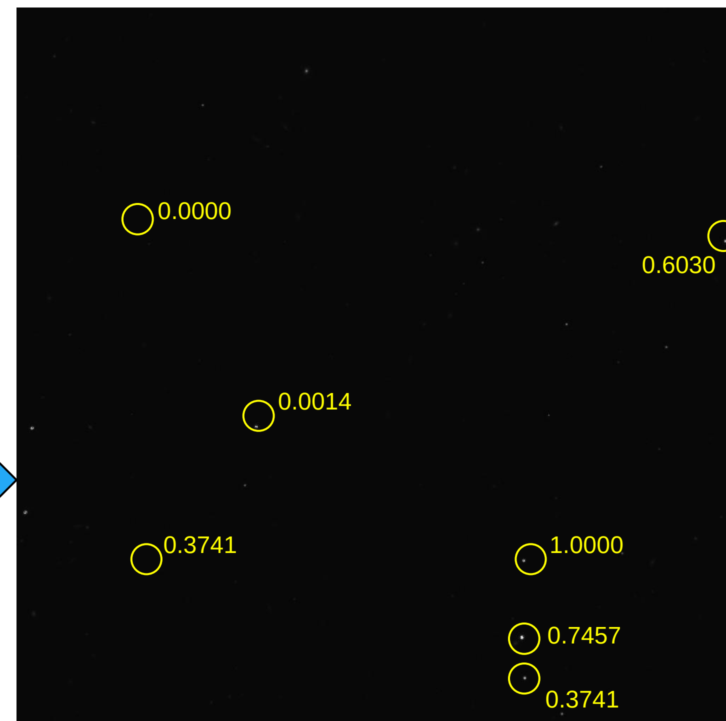
Example: Dinosaurs

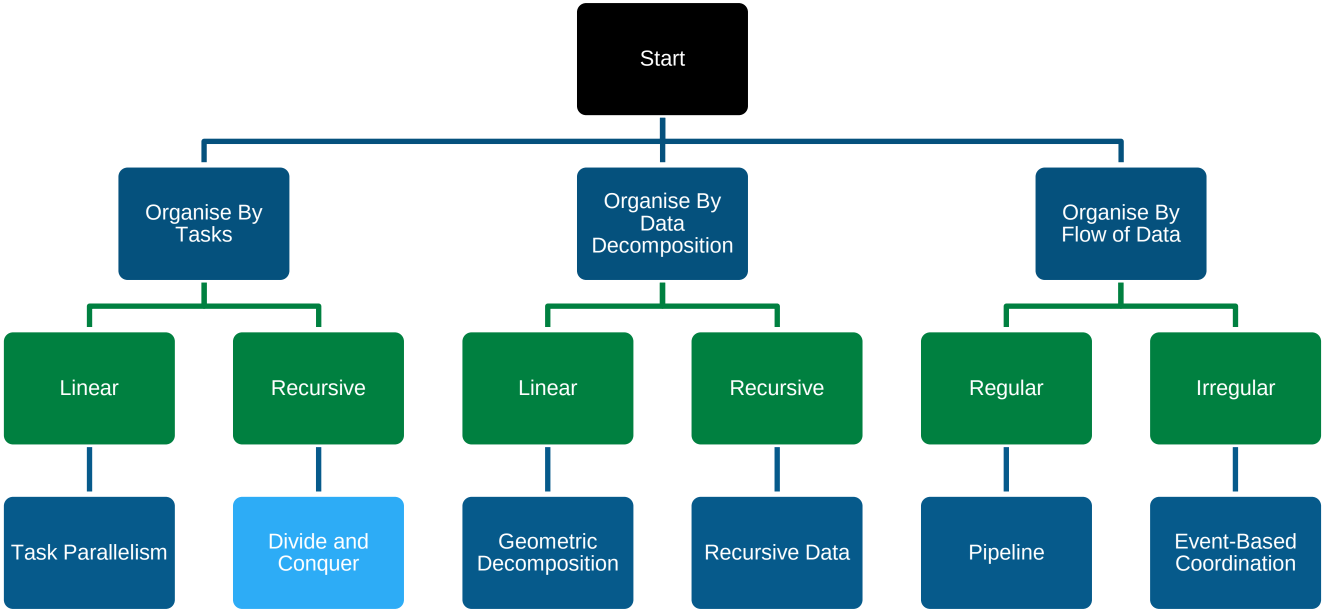


Example: Star Extractor

- 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





Pattern:

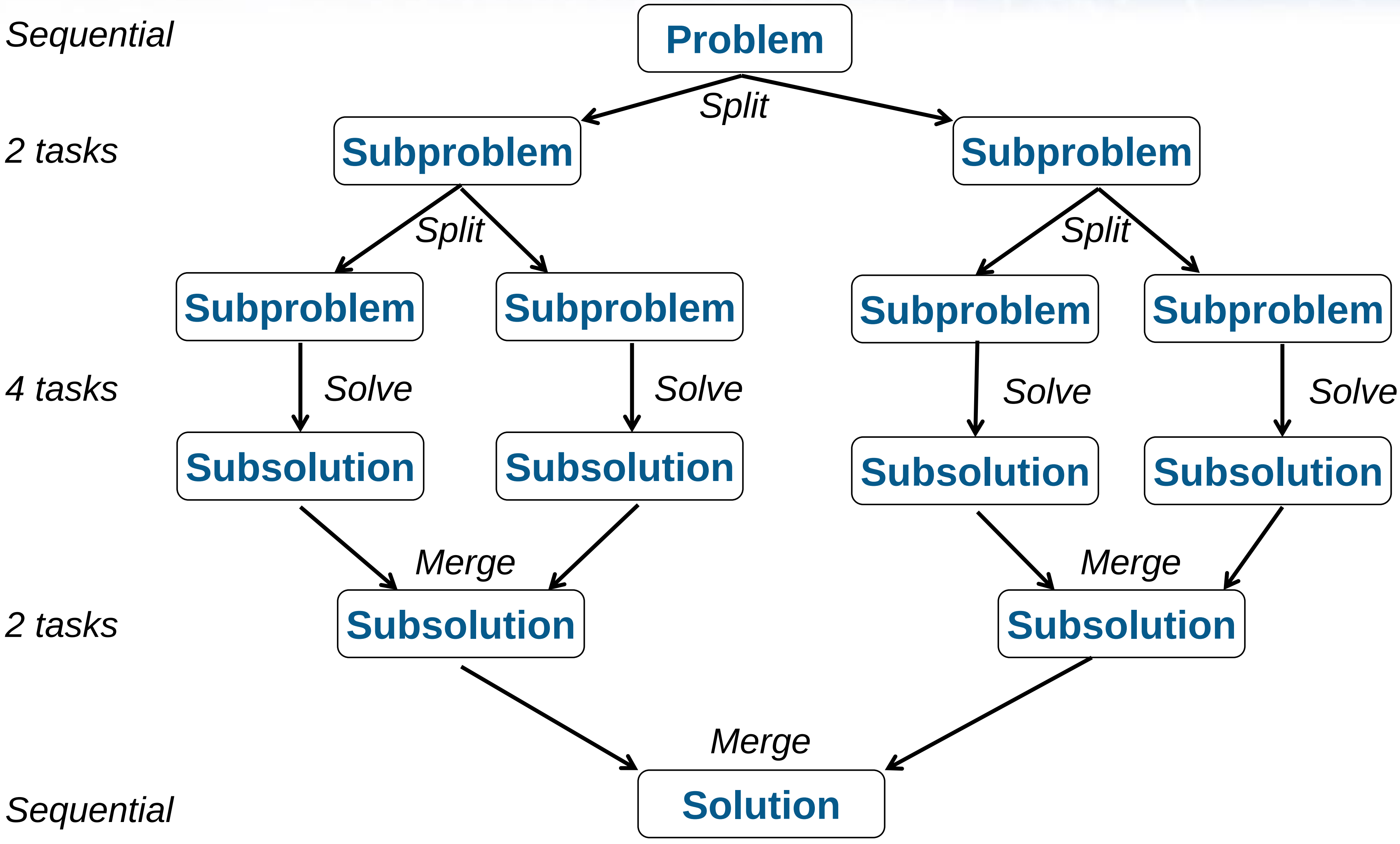
DIVIDE & CONQUER

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 10), or
  - Master/Worker pattern (see Lecture 9)

- **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

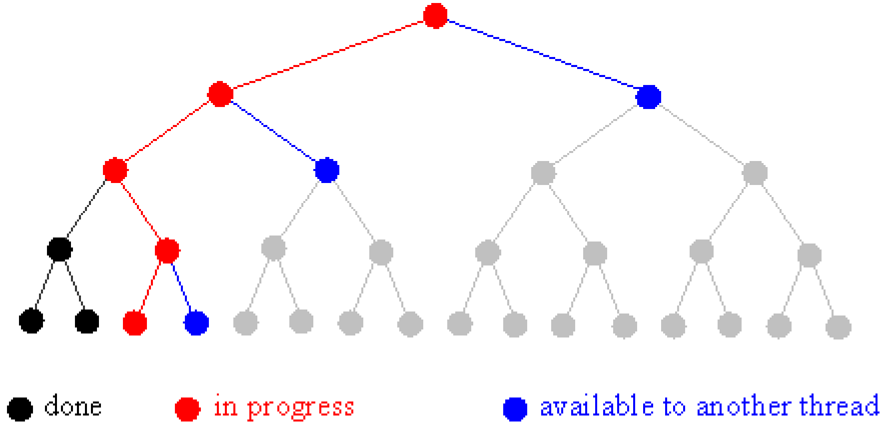
```
sort(int[] A) {
  if (length(A) < THRESHOLD) {
    return quicksort(A)
  } else {
    pivot=length(A)/2;
    t=create new task {
      B=sort(A(1:pivot))
    }
    C=sort(A(pivot:length(A))
    wait for t to complete

    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

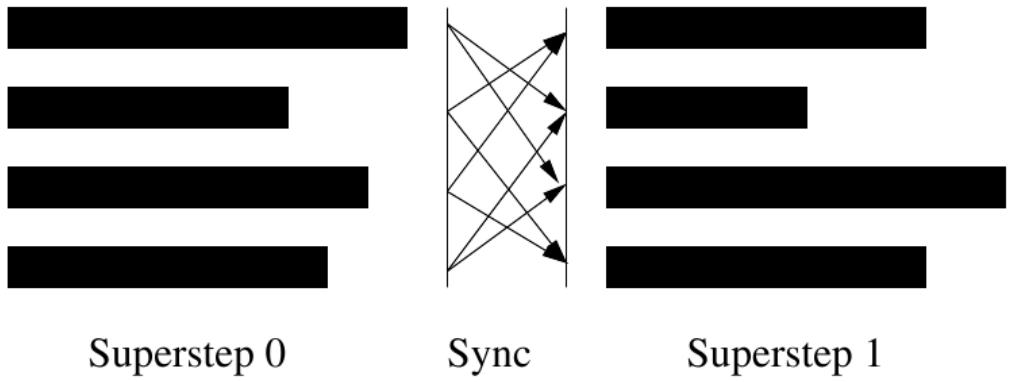


Research space:

RECASTING DATA PARALLELISM INTO (RECURSIVE) TASK PARALLELISM

Moving towards exascale

- Decomposing based on the data has its limits
  - In terms of scalability
  - In terms of resilience
- Parallelism oriented around data (e.g. geometric decomposition) can easily fall into the trap of being Bulk Synchronous
  - Such as a halo swap, where all UEs are involved in the communication (even not directly) means that there is significant synchronisation here



Task based programming can help

- Split your problem up into tasks, tasks might be able to spawn new tasks and have dependencies that must be satisfied before they are run

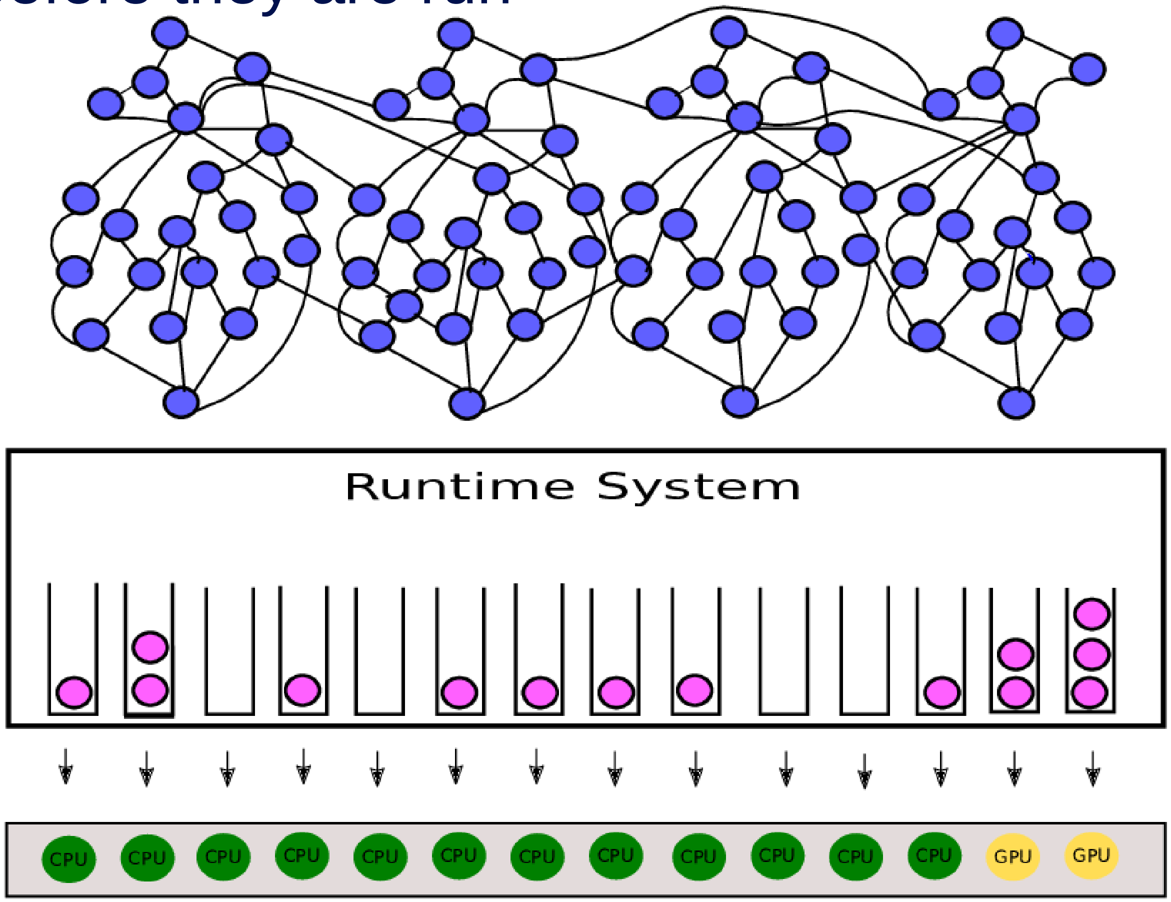


Image taken from http://morse.gforge.inria.fr/chameleon

Why is this useful?

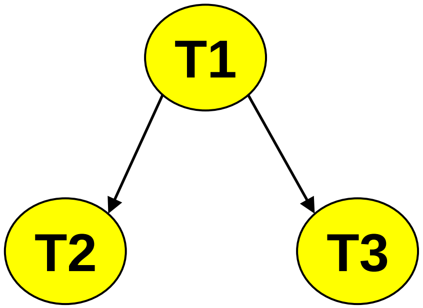
- These tasks are defined asynchronously, so can run entirely independently (no communication during task execution.)
  - Forces the programmer to rethink their problem and break this bulk synchronicity that can be implicit
- Can naturally balance the work by clever scheduling (see slide 15.)
- Can help with resilience, as if a node goes down then the scheduler simply reschedules the task(s) on other nodes
  - Can also run multiple instances of the same task and compare results to ensure no undetected hardware errors
- Architecturally independent
  - More easily take advantage of heterogeneous machines

Task based programming technologies

- OpenMP
  - Task dependencies in OpenMP 4

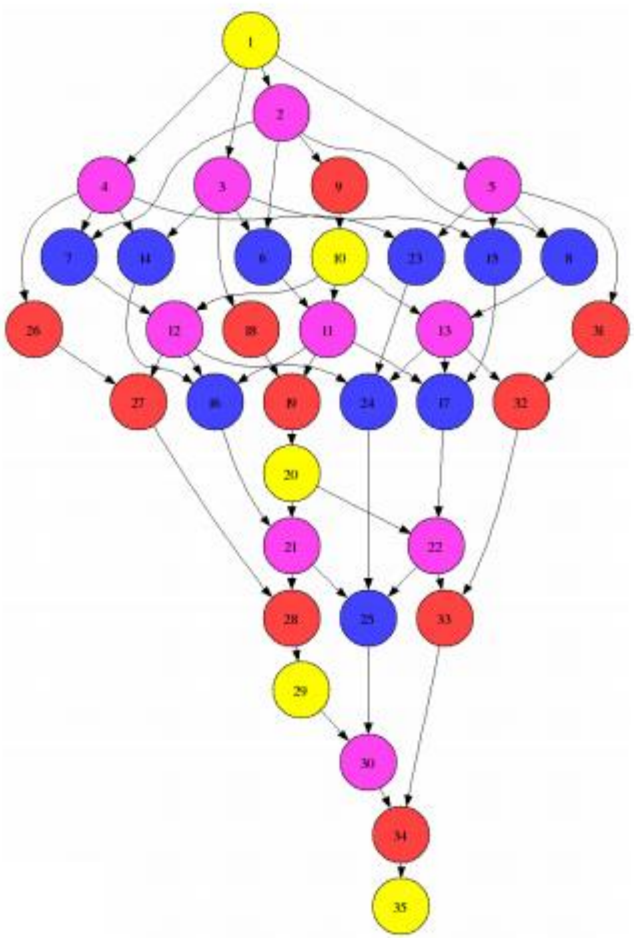
```
void process_in_parallel() {
  #pragma omp parallel
  #pragma omp single {
    int x = 1;
    ...
    for (int i = 0; i < T; ++i) {
      #pragma omp task shared(x, ...) depend(out: x) // T1
        preprocess_some_data(...);
      #pragma omp task shared(x, ...) depend(in: x) // T2
        do_something_with_data(...);
      #pragma omp task shared(x, ...) depend(in: x) // T3
        do_something_independent_with_data(...);
} } }
```

- T1 has to complete before T2 and T3 can execute
- T2 and T3 can run in parallel



- OmpSs, StarSs, StarPU
  - Research languages for tasks based programming
- Legion, OCR, HPX
  - Typically used by limited numbers of organisations

Cholesky Decomposition example



```
for (j = 0; j < NB; j++) {
   #pragma omp task inout(A[j][j])
   spotrf( A[j][j]);

   for (i = j+1; i < NB; i++) {
      #pragma omp task in(A[j][j]) inout(A[i][j]
      strsm( A[j][j], A[i][j]);
   }


   for (k = 0; k < j; k++)
      for (i = j+1; i < NB; i++) {
         #pragma omp task in(A[i][k], A[j][k])
inout(A[i][j]
         sgemm(A[i][k], A[j][k], A[i][j]);
      }

   for (i = 0; i < j; i++) {
       #pragma omp task in(A[j][i]) inout(A[j][j])
       ssyrk( A[j][i], A[j][j]);
   }


}
```

- Example taken from http://www.slideshare.net/IntelITCenter/ompss-improving-the-scalability-of-openmp

Current research challenges

- All these technologies currently have their own compiler technology
- How to recast algorithms into tasks
  - Ensuring performance, the task granularity is important as too many tasks results in significant scheduling overhead
- The cost of parallelism on distributed memory machines
  - How to most effectively do this?
  - Communication (input & output of tasks)
- How do we best express these tasks?
  - Including all the options that go with them
  - The programmer probably doesn’t want to rely on the runtime to make all these choices, so how can we do this easily and transparently?

Conclusions

- We have talked about two patterns which can be used when the concurrency's organising principal is that of the tasks themselves

- Task parallelism where the tasks are linear and created sequentially
- Divide and conquer when the tasks are created recursively

- Current research challenges applying task based parallelism to data centric designs