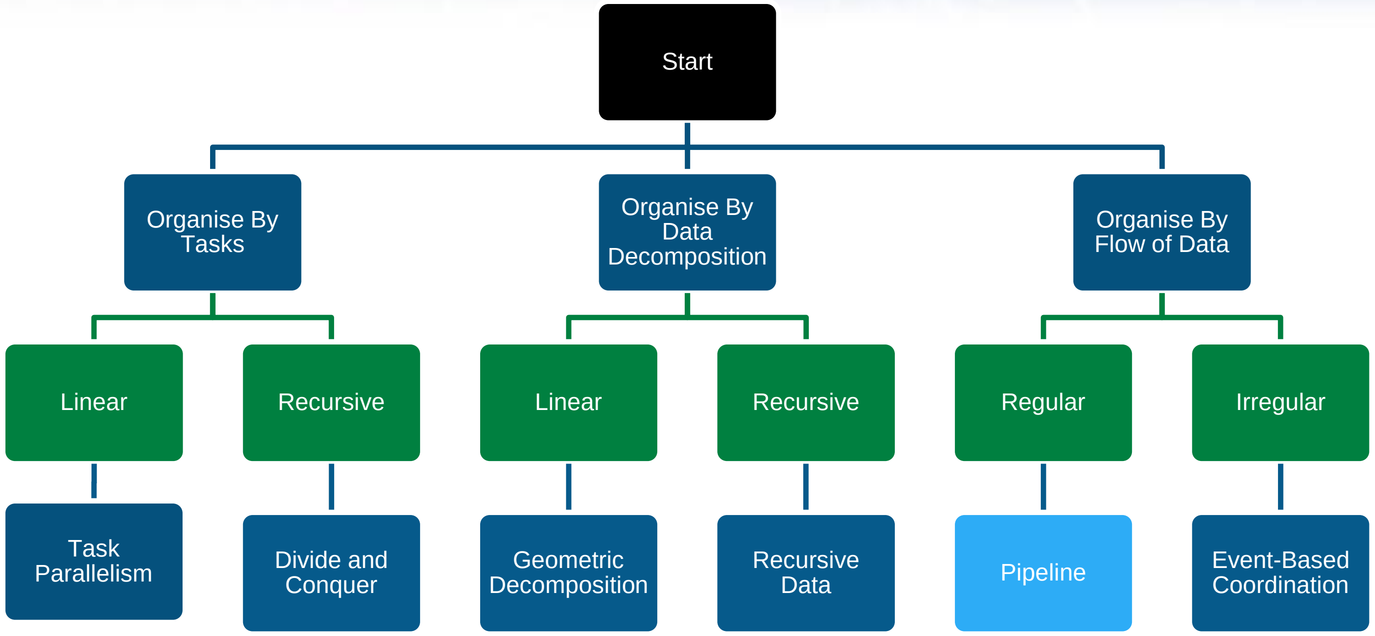
Parallel Design Patterns-L05

Pipelines,

Event Based Coordination

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



PIPELINE PATTERN:

A problem involves operating on many pieces of data in turn. The overall calculation can be viewed as data flowing through a sequence of stages and being operated on at each stage. How can the potential parallelism be exploited?

Pipeline – The Context

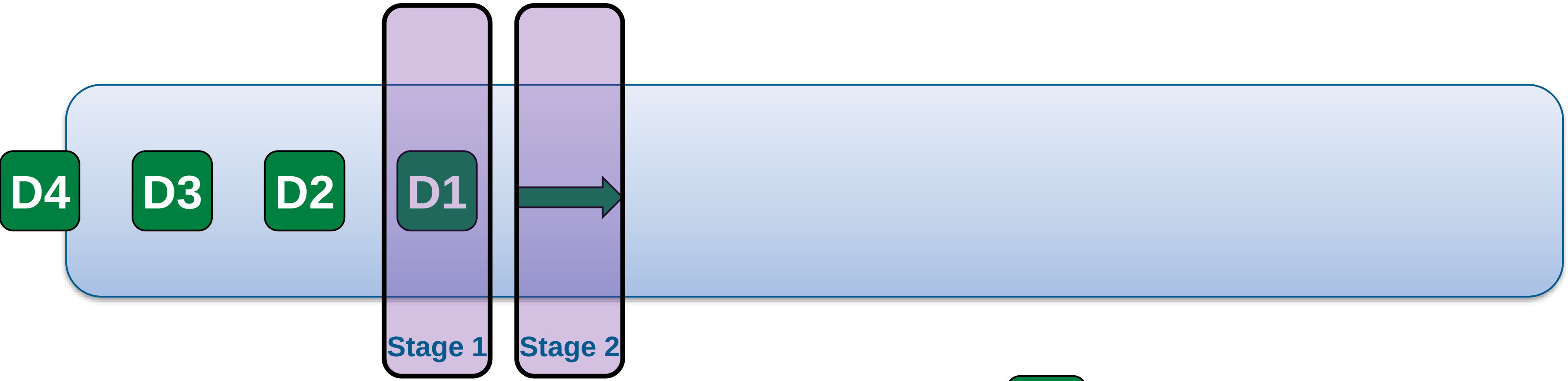
- An assembly line is provides a very good analogy
  - Instead of a partially assembled car, we have data
  - Instead of workers or machines, we have UEs
- Pipelines are found at many levels of granularity
  - Instruction pipelining in CPUs
  - Vector processing (loop-level pipelining)
  - Algorithm-level pipelining
  - Signal processing
  - Shell programs in UNIX
- Pipeline pattern exploits problems involving tasks with straightforward ordering constraints
  - A number of operations working on each data element in sequence
  - There is an ordering constraint for each piece of data, but crucially operations can run concurrently on different data elements

Pipeline – The Forces

- A good solution should make it easy to express ordering constraints
  - These are simple and regular
  - Lend themselves to being expressed as data flowing through a pipe
- Target platform should be borne in mind
  - Sometimes contains special hardware to perform some of this
- In some applications, future modifications to (or reordering of) the pipeline should be expected

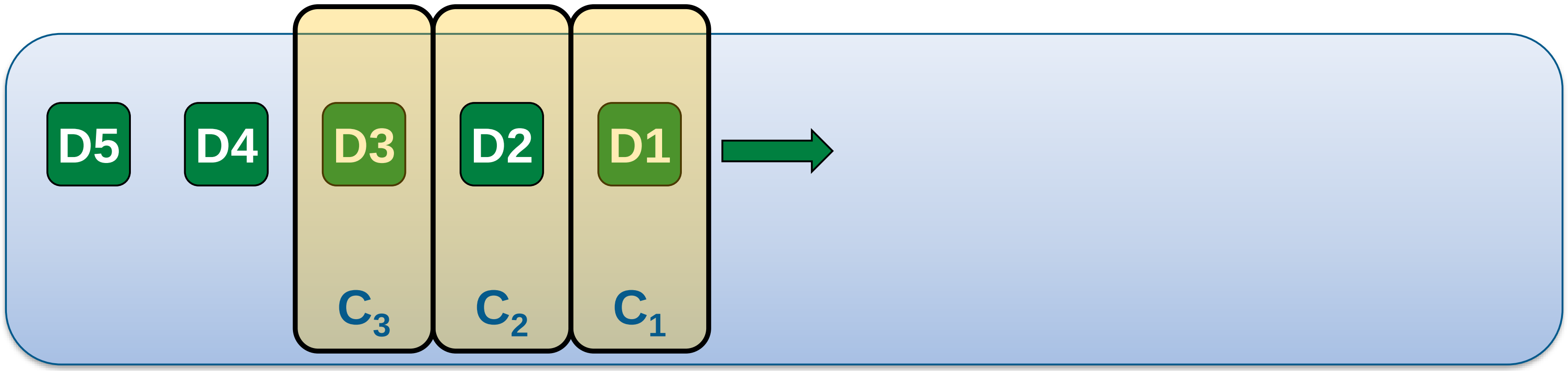
Pipeline – The Solution

- Idea is captured by the assembly line analogy
- Assign each computation stage to a different UE and provide a mechanism to so that each stage of the pipeline can send data elements to the next stage

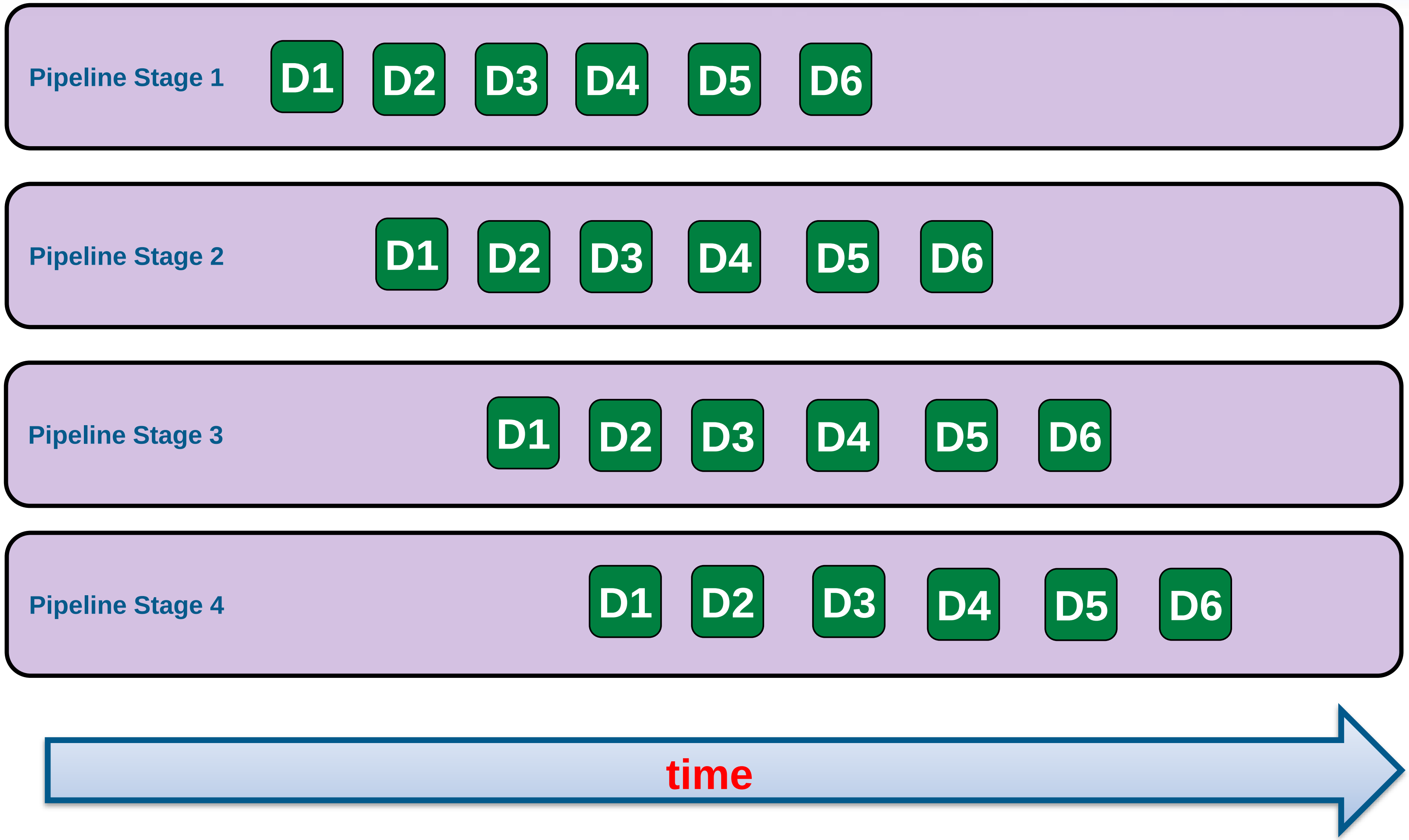


Dn = the *n*th piece of data

Snapshot of pipeline at *t* = 3

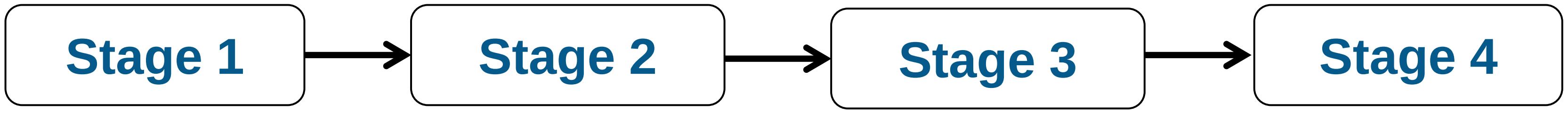


Pipeline example

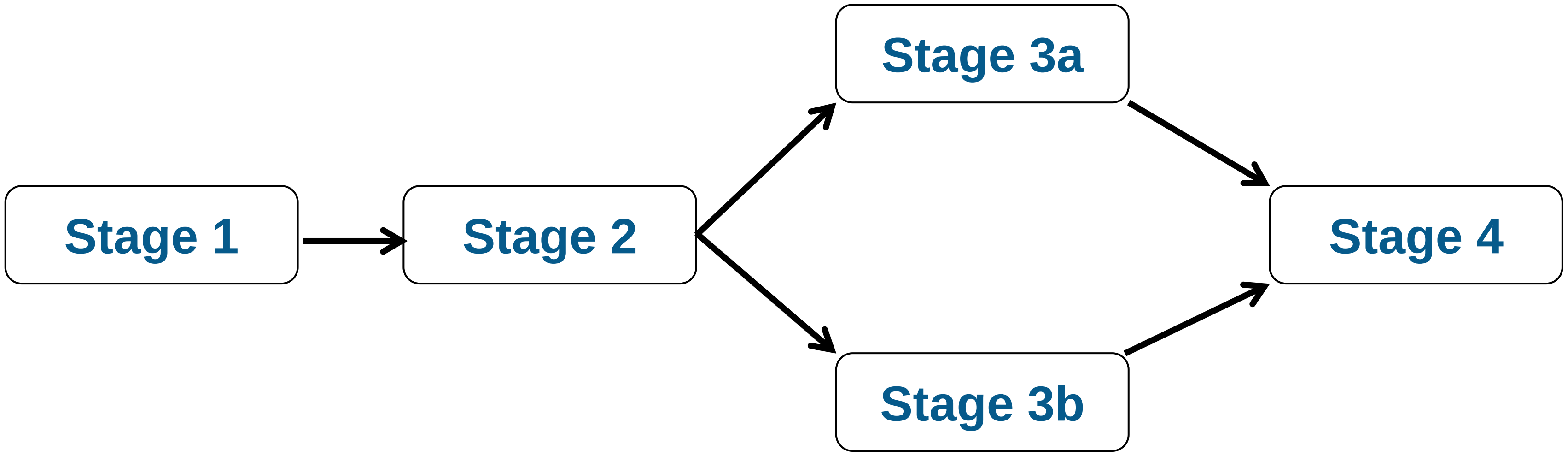


At time=4 the pipeline is filled, starts to drain at time=7

Pipeline architectures



Linear pipeline



Non linear pipeline

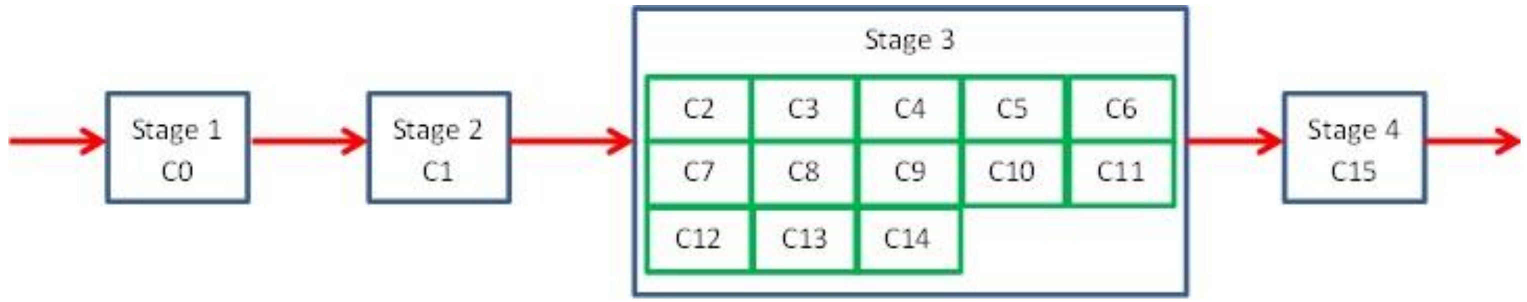
- Throughput (bandwidth) is number of data items per time unit that can be processed (once the pipeline is full.)
- Latency is the amount of time taken for the processing of a single element of data

Defining the stages of the pipeline

- Normally each pipeline stage will correspond to one task
- Pipeline stage shuts down when
  - It has counted that it has completed all tasks (if count is known)
  - Receives a shut-down “sentinel” (poisoned pill) through the pipe
- Concurrency is limited by the number of stages
  - But data must be transferred between stages
- Pattern works best if all stages are of similar computational intensity
- Pattern works best if time to fill/drain pipeline is small compared to the running time
  - or, equivalently, if latency is small compared to bandwidth
  - depends on depth of pipeline, number of data elements

Structuring the computation

- Need to define the overall computation
- Pipeline commonly used with
  - SPMD pattern, using a UE's identifier to differentiate between options in a case statement
- Pattern can be combined with other Algorithm Strategies to help balance load amongst stages
  - e.g. one pipeline stage could be parallelised with *Task Parallelism*



Representing the dataflow

- Driven by the available supporting structures in the language/architecture

- With MPI: Map dataflow between elements to messages
  - Point to point communications
  - One process to each stage in the pipeline

- With Shared Memory: use the *Shared Queue* pattern
  - Which we will see later in the course

Pipeline code sketch

- Each pipeline stage will have the following structure:

```
initialise
while(not done)
{
  blocking receive data from previous stage
  process data
  send processed data onto next stage
}
send termination sentinel to next stage
finalise
```

- The sending of data can be non-blocking or a buffered call
- Your termination sentinel could be an empty message and you could check the number of data elements received

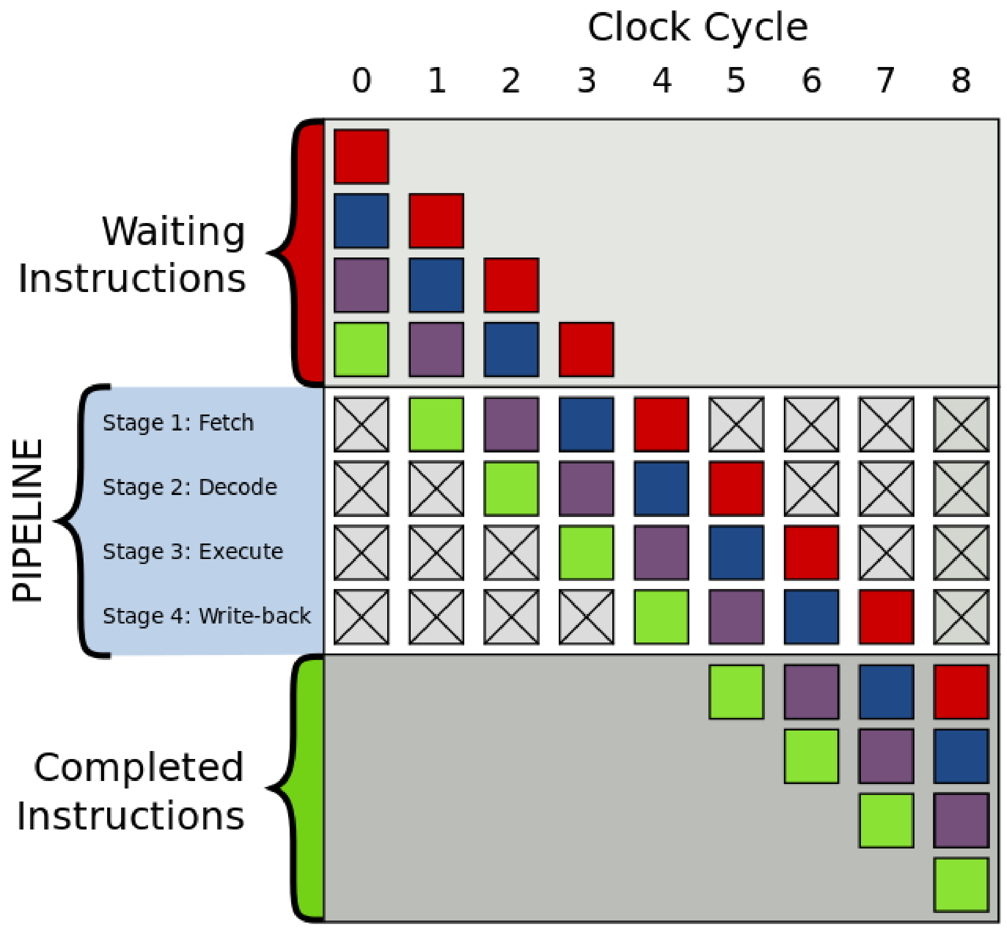
Handling errors

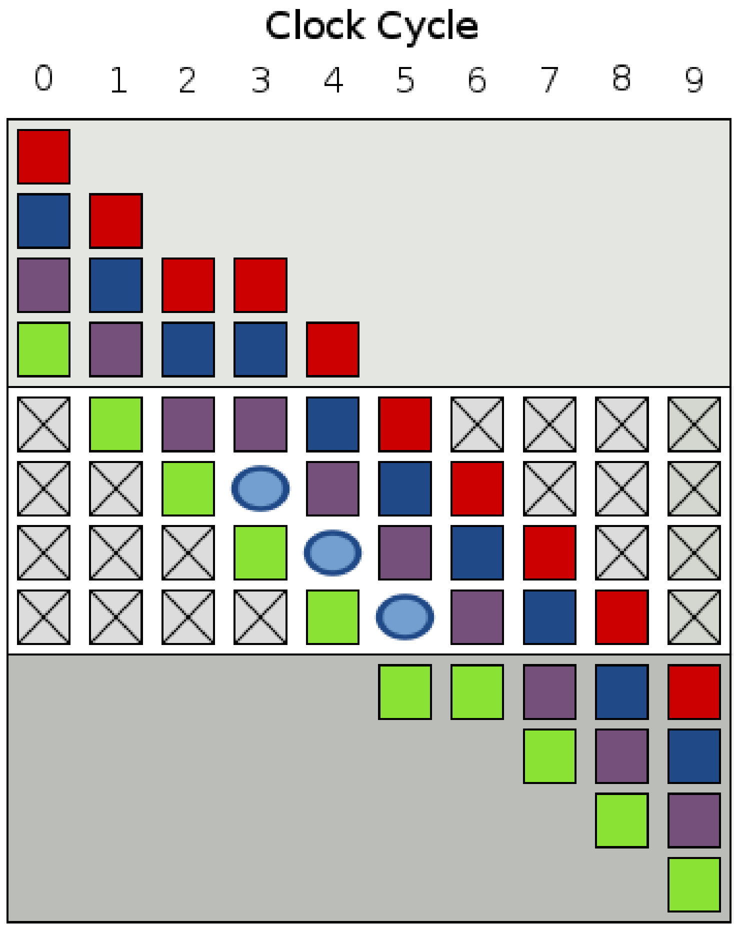
- Potential for “a spanner in the works”
- If there’s an error in one part of the pipeline, it has potential to break the whole flow
  - Do I care? Sometimes no, but sometimes yes – especially if you need a 1:1 correspondence between input tokens and output tokens

- Common solution is to have a separate “error handling” task (or tasks) with which pipeline elements can communicate
  - Keep pipeline flowing
    - Pass an “error” sentinel, often known as a *bubble*
    - Acts as a “noop” for each subsequent stage

Instruction Pipelines

- 4 activities/instruction: Fetch, Decode, Execute, Write-back





Processor allocation & task scheduling

- Simplest approach: One UE per pipeline stage
  - Good load balance if the amount of work per pipeline stage is roughly the same

- If there are fewer UEs than pipeline stages:
  - Oversubscribe: Assign multiple stages to the same UE. Ideally those that do not share resources. By assigning neighbouring stages to a single UE can reduce communication overhead, or
  - Combine stages into a single bigger stage

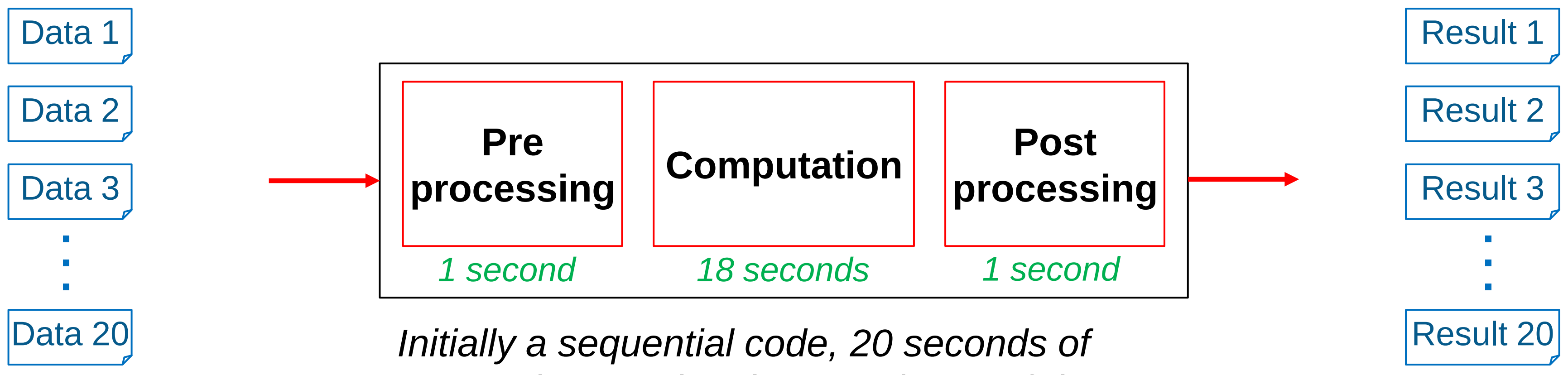
- If there are more UEs than pipeline stages:
  - Parallelise a stage (task parallelism within pipeline), or
  - Run multiple pipelines (pipeline within task parallelism) as long as there are no temporal constraints between the data

UNIX instruction pipeline

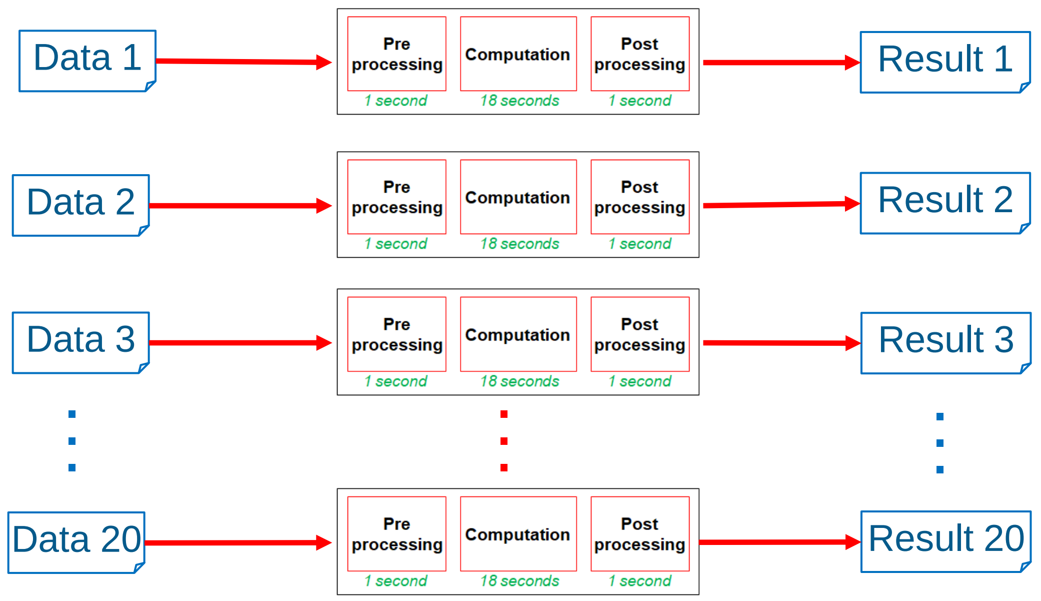
```
cat datafile | grep “energy” | awk ‘{print $2, $3}’
```

- Implemented by starting three processes and using buffers implemented using *shared queues*
- Processes are connected by their **stdin** and **stdout** streams
- Implemented as an anonymous pipe (which breaks as soon as processes complete)
- UNIX also provide named pipes which can persist between program invocations
  - Look like files
  - Created with `mkfifo` command

Computation example



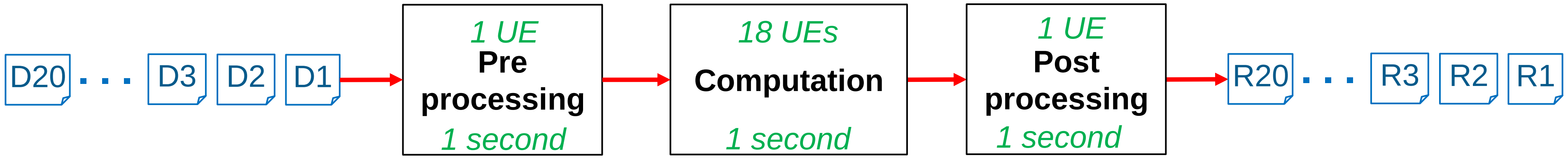
*Initially a sequential code, 20 seconds of processing per data item, 20 items of data: 20*20 = 400 seconds runtime*



- *Parallelised this by splitting up into 20 separate tasks, one task per element of data*
    - *Run over 20 UEs*
- *Overall runtime is now 20 secs*

- *But no results are generated until 20 seconds, at which point all results are available*

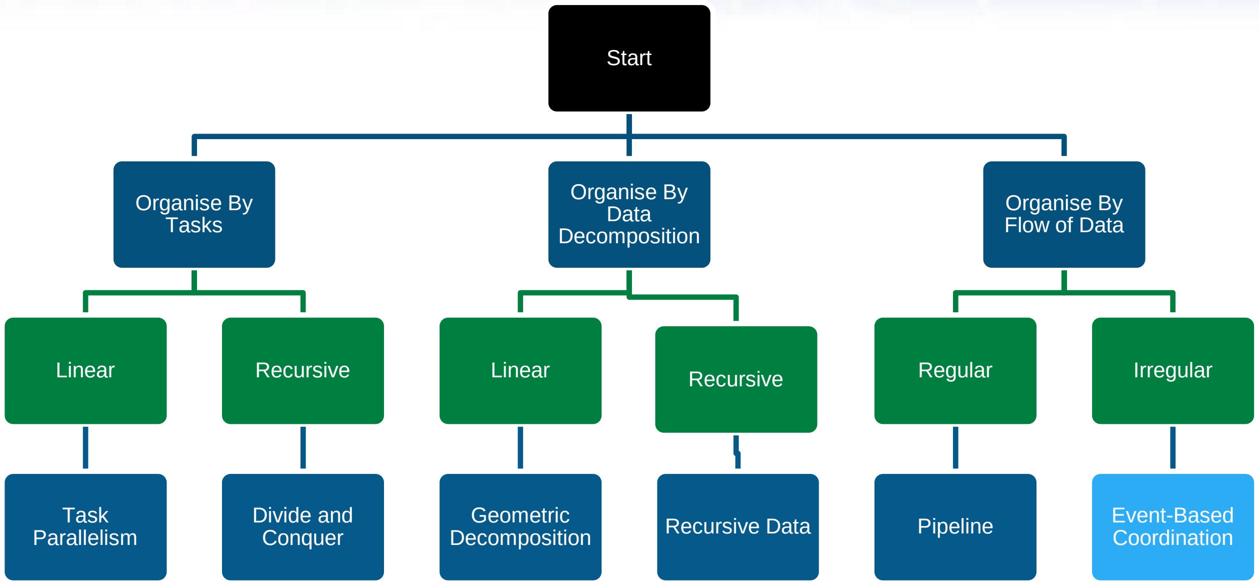
Computation example



- Using a parallel pattern (e.g. geometric decomposition) allocate 18 UEs to the computation stage, 1 UE to other two stages.
- Therefore each stage now takes 1 second = 3 seconds in total
- If we don't run as a pipeline
  - Only start to process subsequent element of data when preceding element is fully processed
  - Takes 3 seconds per element to be processed (a result is generated every 3 seconds)
  - 3 * 20 = 60 seconds total runtime
- If we run as a pipeline
  - Each stage runs concurrently, processing a different piece of data
    - Still takes 3 seconds to process first element of data, but once this is done and the pipeline is fully filled then generate a result every second
    - 3 + 19 = 22 seconds total runtime
    - Slightly longer than have a separate task work on every element of data, but crucially once the pipeline is filled we are generating a result every second

Pipelines: Conclusion

- Pipelines exist at various different levels in software and hardware
  - Forms a corner stone of processor microarchitecture design
- The pattern is also useful more generally and often used together with other patterns for applications characterised by a regular flow of data
  - For instance for data pre and post processing
- A generalisation of Pipeline (a workflow, or dataflow) is becoming more and more relevant to large, distributed scientific workflows



EVENT BASED COORDINATION PATTERN:

The Problem: An application can be decomposed into groups of semi-independent tasks interacting in an irregular fashion. The interaction is determined by the flow of data, implying ordering constraints. How can the tasks and their interactions be arranged so that they can run concurrently?

Event-Based Coordination: Context

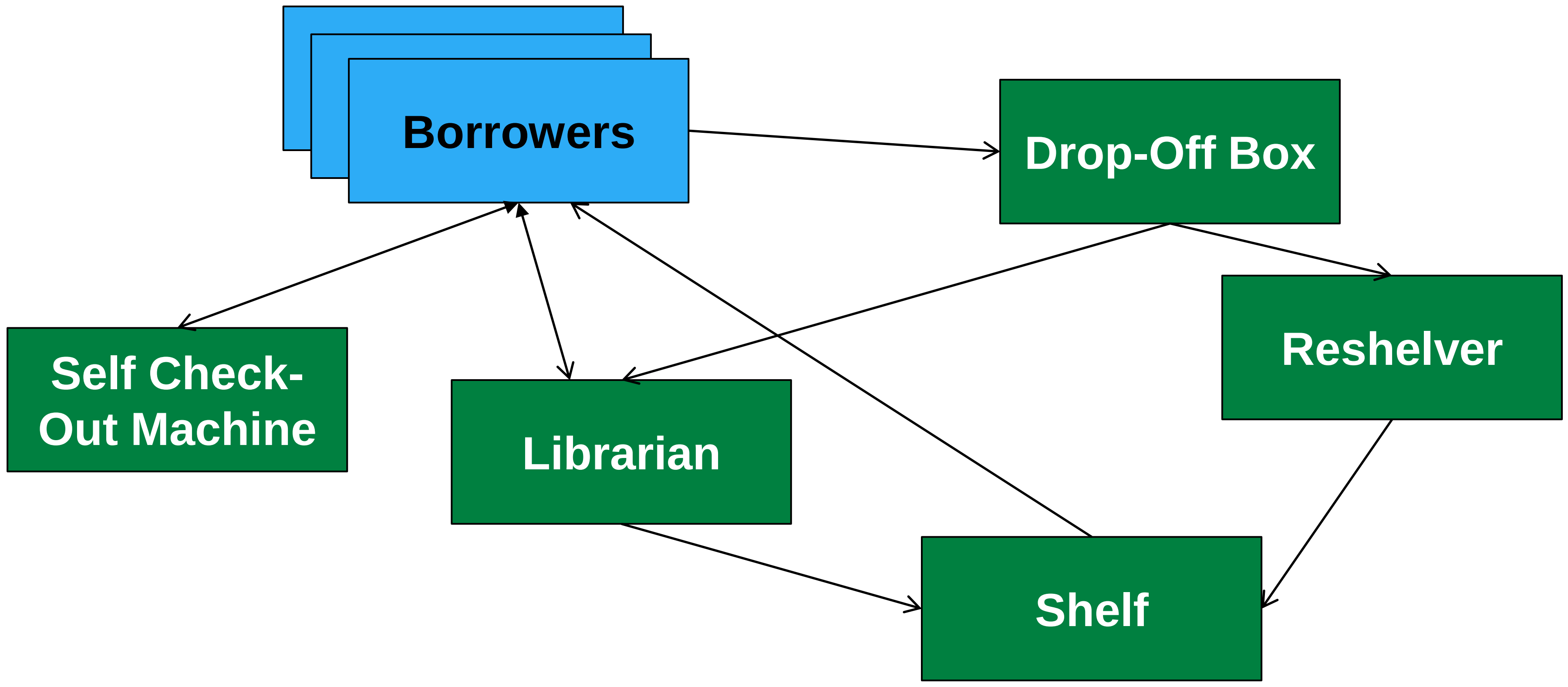
- Semi independent entities which are interacting in an irregular fashion
  - Events sent from one task to another and processed by a *handler*

- In contrast to *Pipeline:*
  - No restriction to a linear flow of data
  - Might communicate with any other task
  - Interaction can take place at irregular (and unpredictable) intervals
  - Different events (data) can be handled in very different ways

- Closely related to discrete-event simulation
  - Simulations of real-world processes, in which real-world entities are modeled by objects that interact through events

Discrete-Element Simulation

- Example: Modeling the flow of books in a library:



Crucially, each stage is fundamentally still responding to some data (event) arriving at it and does nothing otherwise

Event-Driven Applications

- Not just for simulations
- Pattern can be applied to real-time applications
  - e.g. Monitoring / controlling systems in a power station
- Most GUI-based applications are event-driven
  - Events come from user (keyboard, mouse, etc) and system
  - Not massively parallel, but can benefit from parallelism
- Distributed applications
  - Events come over the network
  - e.g. Google Docs
    - More complex then you might think!

Event-Based Co-ordination: Forces

- A good solution should
  - Make it easy to express ordering constraints
    - possibly numerous, irregular, arising dynamically
  - Expose as much parallelism as possible
    - By allowing as many possible concurrent activities as possible

- This solution should provide an alternative to trying to express constraints in other ways such as might be found with other patterns, such as
  - Sequential composition
  - Shared variables representing state

Event-Based Co-ordination: Solution

- Events – sent from one task (source) to another (sink)
  - Implies an ordering constraint
  - Computation consists of processing events
- One task per real-world entity
  - And usually one UE per task
- Elements of the solution
  - Defining the tasks
  - Representing event flow
  - Enforcing event ordering
  - Avoiding deadlocks
  - Scheduling processor allocation
  - Efficient communication of events

Defining the Tasks

- Each task will have the following structure:

```
initialise
while(not done)
{
  receive event
  process event
  send events
}
finalise
```

- If program is being composed from existing components, these can be "wrapped" to give an event based interface
  - This is an example of the *Façade* pattern (GoF)

Representing Event Flow

- To allow communication and computation to overlap (and generally to avoid serialisation) you need a way to communicate events asynchronously

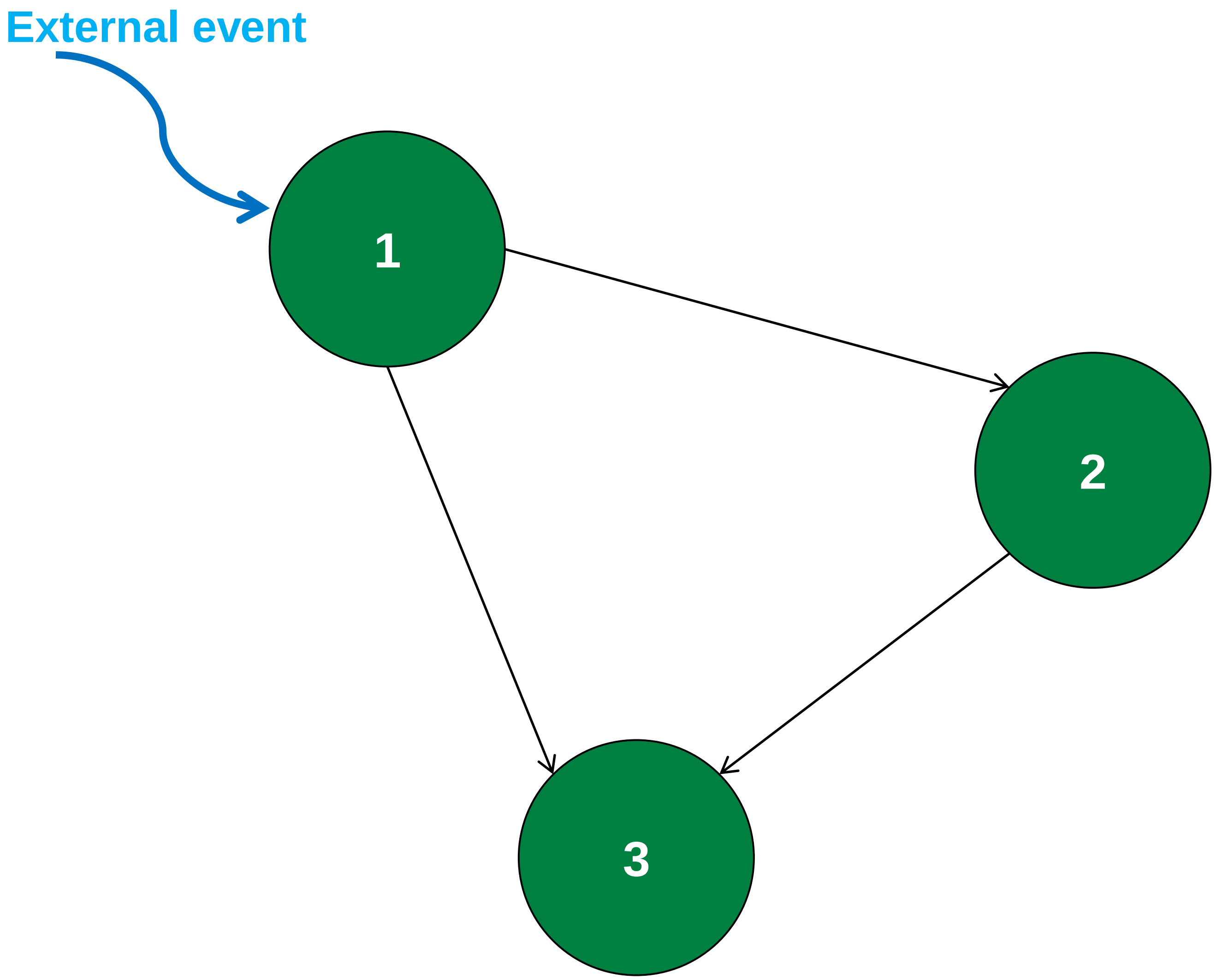
- In a message passing environment, the event can be represented by a message sent asynchronously from the task that generated it to the task that is to process it

- In a shared memory environment, a shared queue can be used to simulate message passing
  - We will cover this pattern later in the course

Enforcing Event Ordering

- Probably the hardest step in applying this pattern
- Enforcing ordering constraints might make it necessary for a task to *process* events in a different order from which they are *sent*
  - Or, indeed, *received*. Note therefore that MPI’s guaranteed P2P message ordering is not necessarily enough to protect you here!
  - *Events* are often received from different UEs which can arrive in any order so need to be aware of any constrains here too.
- It is therefore sometimes necessary, depending on the approach taken, to “look ahead” in an event queue to determine what the correct behaviour should be

Event Ordering: Simple Example



Event Ordering

- Before you spend time trying to enforce event ordering, check whether it matters
  - Is event path linear?
  - Does the application care if events are out of order?
- If it does matter, two classes of approach:
  - Optimistic
    - Proceed, and deal with problems later, e.g. by rolling back
  - Pessimistic
    - Wait (e.g. for a synchronisation event) to ensure that ordering constraint is met *but this induces an overhead*
- Mechanisms that can help with event ordering constraints
  - Global clocks
  - Synchronisation events

Avoiding Deadlocks

- Quite a common pitfall with this pattern

- You can get the normal message passing deadlocks, but with event-driven simulation you can also have application-level deadlocks
  - Could be implementation error, or a “model error”
- With pessimistic approach to event ordering, deadlock can arise from being overly pessimistic
  - Some approaches use run-time deadlock detection
    - Often quite inefficient as a general solution
    - Timeouts can be a compromise to full deadlock detection
      - (can be handled locally)

Scheduling and Processor Allocation

- Most straightforward approach is one per UE
  - Allows all tasks to execute concurrently
- Can have several tasks per UE
  - Can be suboptimal in terms of efficiency, but often difficult to avoid
- Load balancing with this pattern can be difficult
  - Infrastructures that support task migration can assist

Efficient Communication of Events

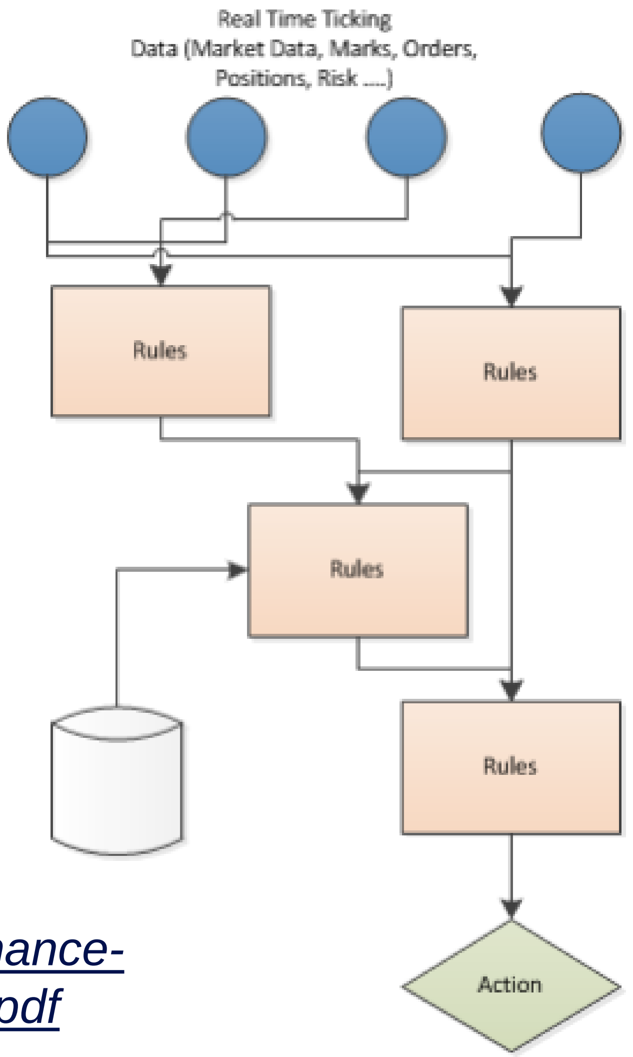
- This model implies considerable communication
- You need an efficient underlying communication system
  - whether it's message-based or shared-memory based
- With a message passing environment it may be possible to combine messages (or sometimes split) to improve efficiency
- With a shared-memory environment, care must be taken that shared queues, etc. are implemented efficiently, as these can easily become bottlenecks

Event-Based Coordination: Summary

- For problems characterised by irregular flow of data
  - Unpredictability of interaction
- Map real-world entities to tasks
- Model real-world interactions with events
- Hardest part to get right is often the event ordering, because communication is not instantaneous, whereas the real-world interactions you're modelling often are synchronous
- Can be flexible in terms of changing details of your model, without changing parallelisation strategy/code
- Can be applied to existing code components

UBS Financial Information Exchange

- Real time market data
  - For quotes, orders & executions
  - Can peak at 16 million items per second (bandwidth)
  - Need to process events in milliseconds (latency)
- Stages operate concurrently
  - Often each run in a separate thread on custom hardware
  - Talk about putting “compute in the data plane”
- Low level optimisation
  - At networking, OS and runtime level
  - FPGA based hardware


Real Time Ticking Data (Market Data, Marks, Orders, Positions, Risk ....)
Rules
Rules
Rules
Rules
Action

http://www.bcs.org/upload/pdf/application-of-high-performance-and-low-latency-computing-in-investment-banks-080115.pdf

https://www.sas.com/content/dam/SAS/en_gb/doc/other1/events/sasforum/slides/manchester-day2/P.Pugh-Jones%20SAS%20Forum%20UK%202015_PPJ_proper.pdf