Parallel Design Patterns

Pipelines and Event Based Coordination

Reusing this material

Pipelines – Problem

- A problem involves operating on a sequence of data items.
- 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?

Pipelines – Introduction

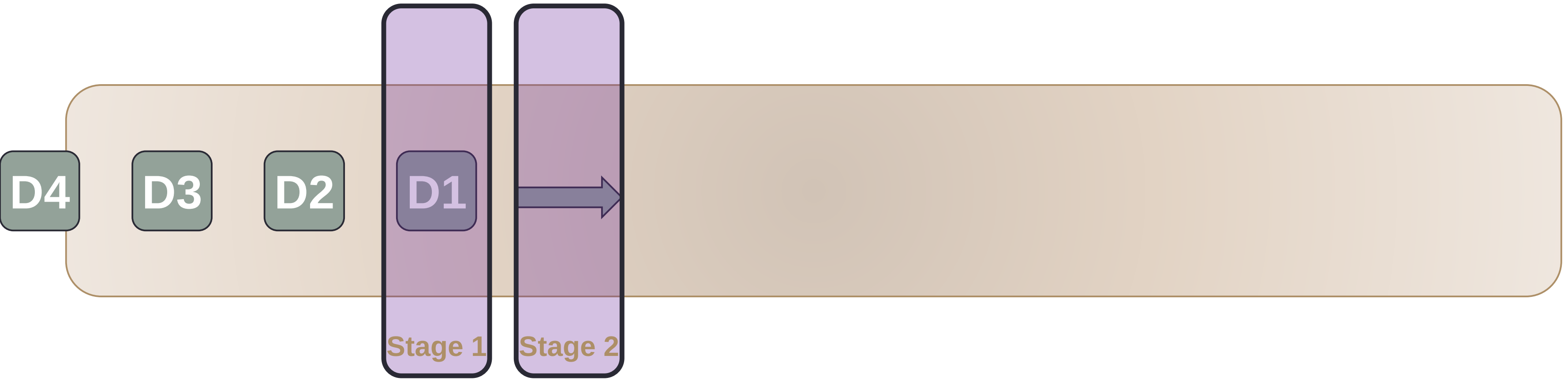
- An assembly line is provides a very good analogy.
  - instead of a partially assembled car, we have data
  - instead of workers or machines, we have UEs
- Pipelines are found at many levels of granularity.
  - instruction pipelining in CPUs
  - signal processing
  - graphics
  - shell programs in UNIX
- The pipeline pattern exploits problems involving tasks with straightforward ordering constraints.

Pipelines – The Forces

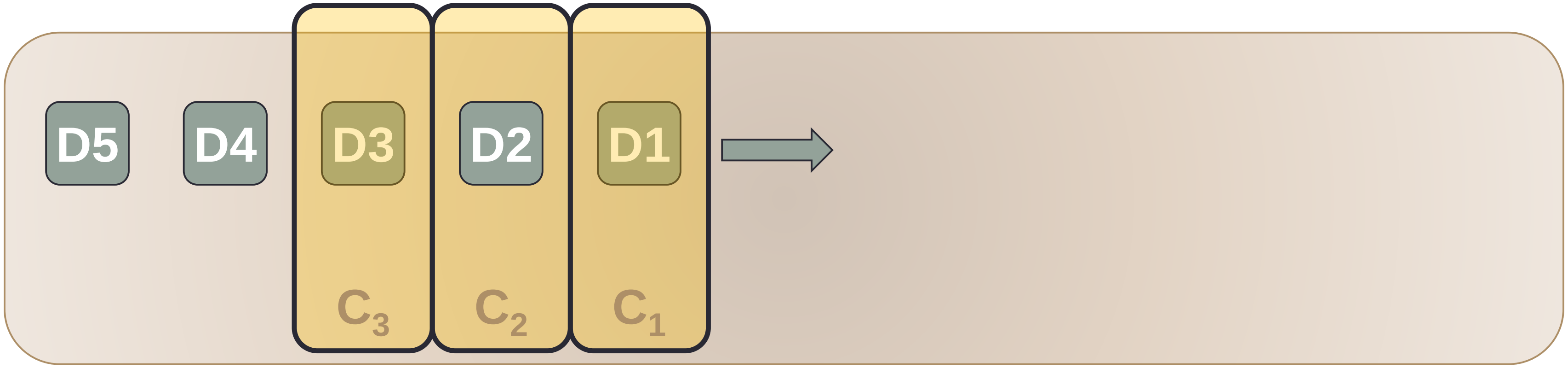
- A good solution should make it easy to express ordering constraints.
  - should be simple and regular
  - should be compatible with the concept of data flowing through a pipe
- Target platform should be borne in mind.
  - it might be possible to implement a pipe stage in hardware
- In some applications, future modifications to (or reordering of) the pipeline should be anticipated.

Pipelines – The Solution

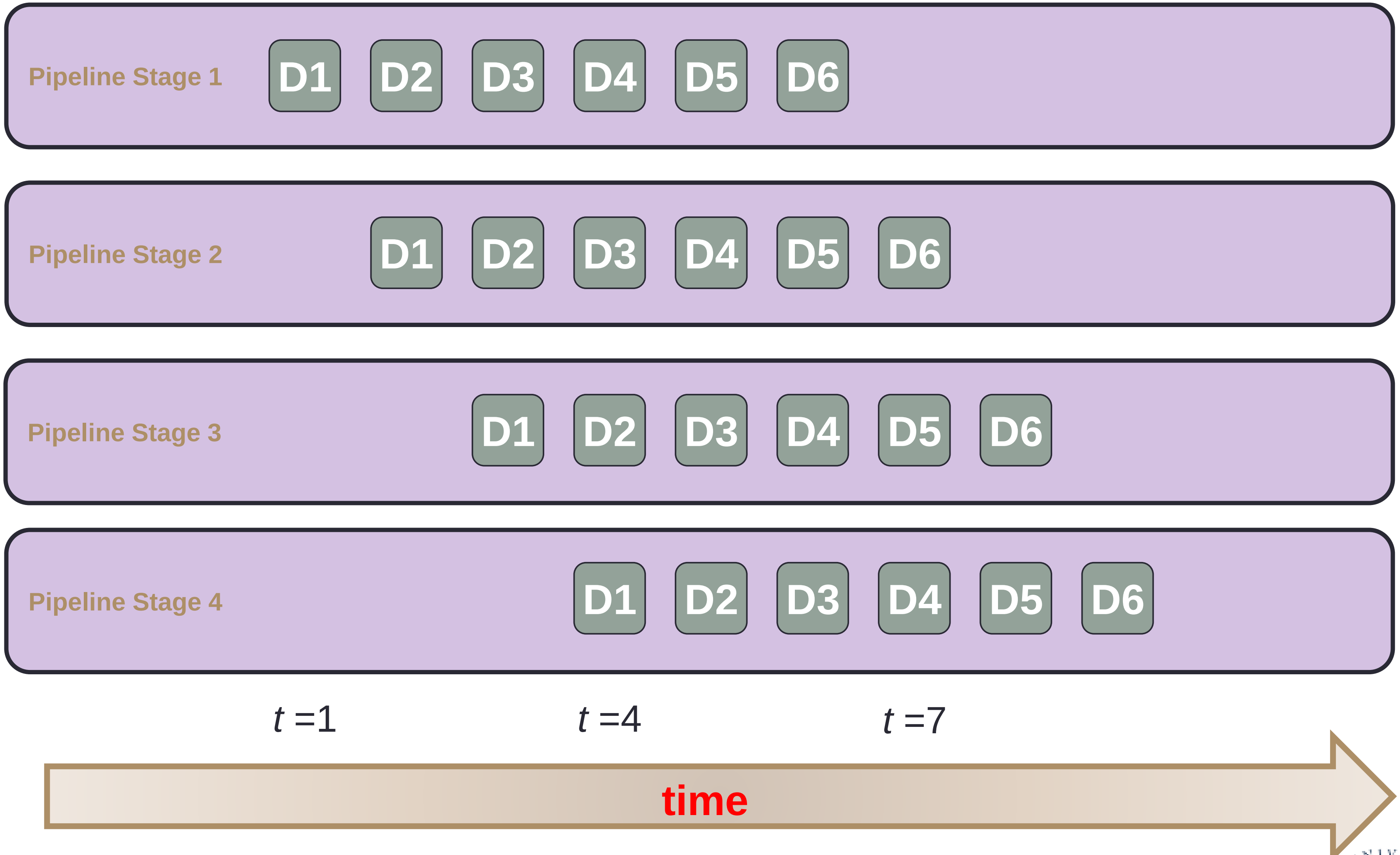
- Idea is captured by the assembly line analogy.
- Assign each computation stage to a different UE and provide a mechanism so that each stage of the pipeline can send data elements to the next stage.



Snapshot of pipeline at $t = 3$

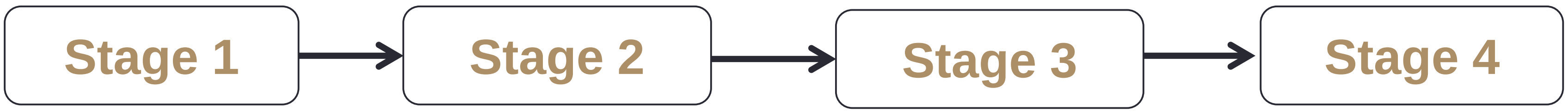


Flow through pipeline

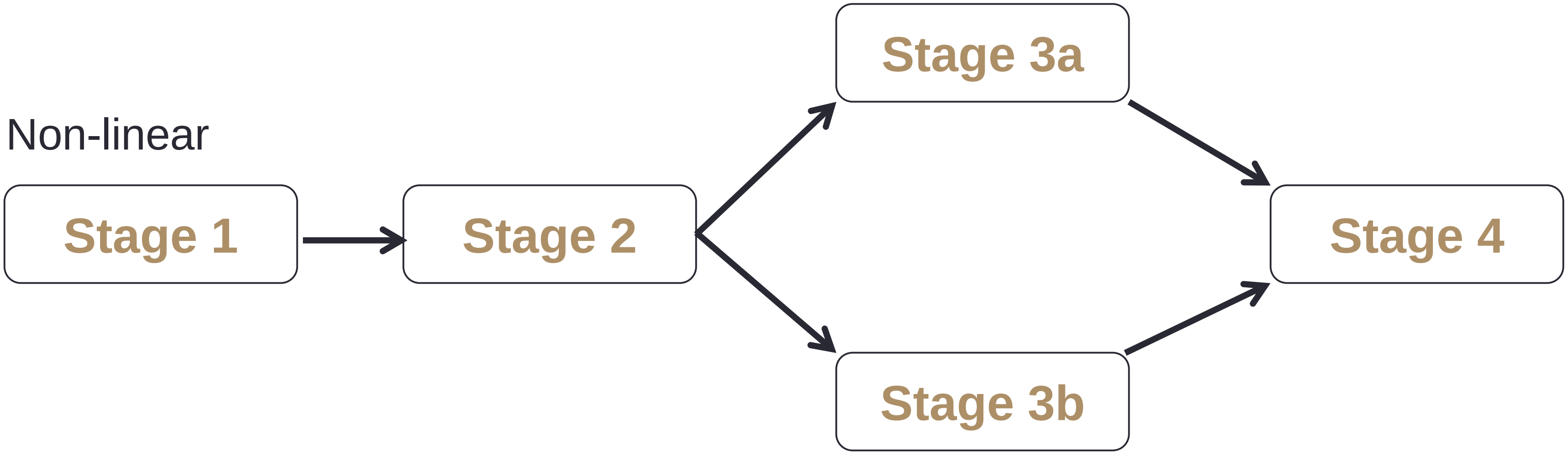


Pipeline architectures

Linear



Non-linear



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)
  - or it receives a shut-down “sentinel” (poisoned pill) through the pipe
- Concurrency is limited by the number of stages.
  - data must be transferred between stages
- Pattern works best if...
  - runtime per stage is constant
  - runtime is large compared to time spent filling/draining pipeline
  - or, equivalently, latency is small compared to bandwidth
    - depends on pipeline length and number of data elements

Structuring the computation

- First define the overall computation - this aspect of the solution is driven by the Implementation Strategy.

- Pipeline commonly used with SPMD pattern.
  - using a UE’s identifier to differentiate between options in a case statement, where each option is a pipeline stage

- 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.

- MPI: map dataflow between elements to messages.
  - one process is mapped to each stage in the pipeline

- Shared Memory: use the Shared Queue pattern.

Pipeline code sketch

- Each pipeline stage will have the following code structure.

```
initialise
while (not done) {
  block 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 (i.e. a buffered call).
- Your termination sentinel could be an empty message or you could check the number of data elements received.

Handling errors

- Obvious 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.

- Common solution is to have a separate "*error handling*" task (or tasks) with which pipeline elements can communicate.
- Important to keep the pipeline flowing.
  - pass an error sentinel or a "noop" result (like a NaN)
  - implementation can depend on whether you need a 1:1 correspondence between input tokens and output tokens

Processor allocation & task scheduling

- The simplest approach is one PE per pipeline stage.
  - good load balance if the PEs are similar and the amount of work per pipeline stage is roughly the same
- What if there are *fewer* PEs than pipeline stages.
  - combine stages into a single bigger stage
  - or, assign neighbouring stages to the same PE
    - reduces communication overhead
    - ideally, stages do not share resources
- What if there are *more* PEs than pipeline stages.
  - parallelise a stage (add task parallelism within pipeline)
  - or, run multiple pipelines (pipeline within task parallelism)
    - fine, as long as there are no reduction constraints on data

epcc

THE UNIVERSITY OF EDINBURGH

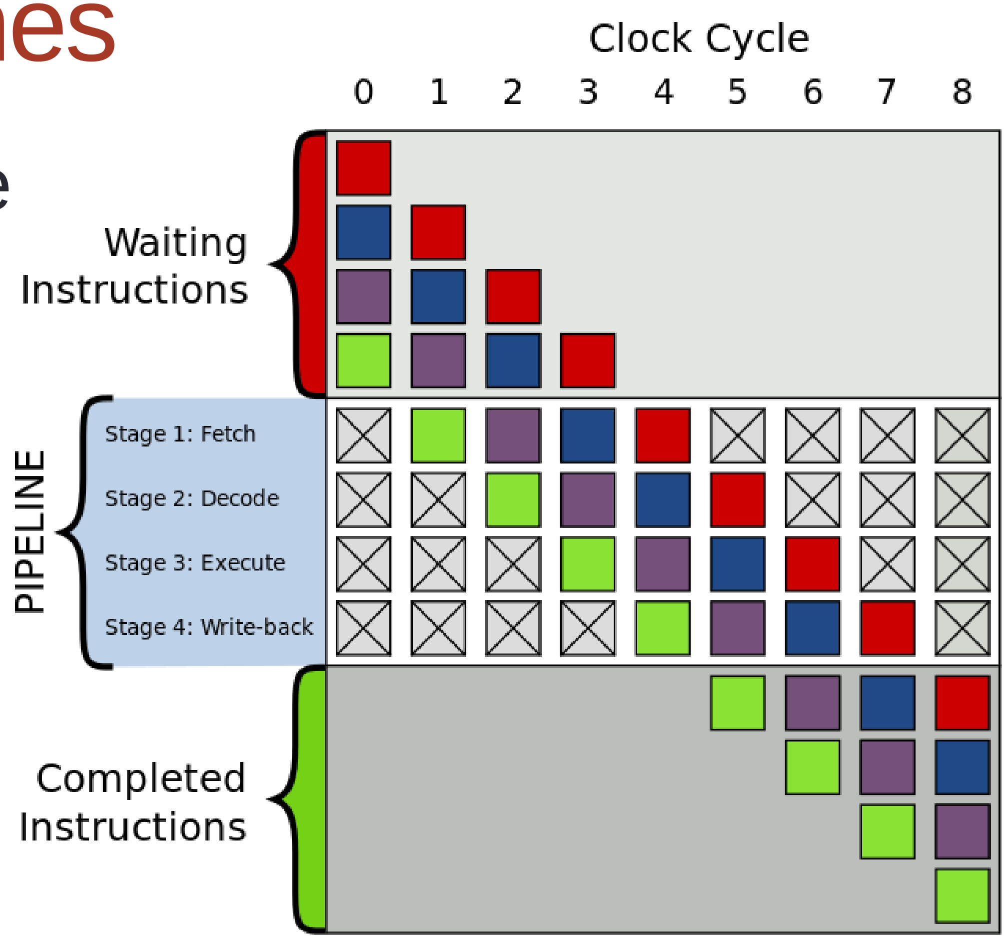
Throughput and latency

- Usually throughput is the most important.
  - number of data items per time unit that can be processed once the pipeline is full

- With small sets of data, or for real-time processing such as live video processing, latency becomes significant.
  - limits length of pipeline

Instruction pipelines

- Fetch | Decode | Execute
  - although, Intel Pentium 4 had 20-stage pipeline

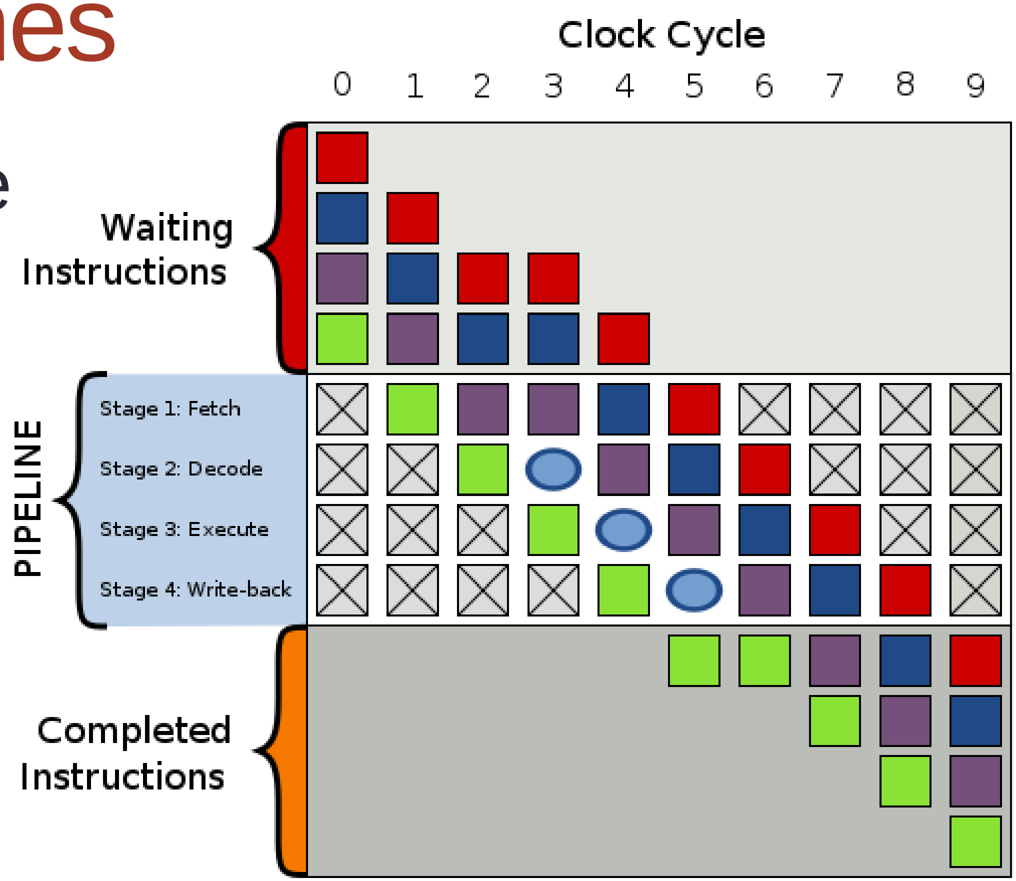


A bubble in an instruction pipeline

Courtesy of Wikipedia, 2011

Instruction pipelines

- Fetch | Decode | Execute
  - although, Intel Pentium 4 had 20-stage pipeline



A bubble in an instruction pipeline

Courtesy of Wikipedia, 2011

UNIX instruction pipeline

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

- Starts three processes and uses buffers implemented as shared queues.
- Processes are connected by their `stdin` and `stdout` streams.
- Multiple-part command is implemented as an anonymous pipe.
  - breaks as soon as processes complete

- UNIX also provides named pipes which can persist between program invocations.
  - persistent pipes created with `mkfifo` command and handled like files

Graphics pipelines

- Hardware (GPUs)
  - ROP (raster operation) pixel pipelines
    - between memory and compute-core in NVIDIA GPUs
  - pipeline hardware can be thought of as special caches
    - that apply transformations to data in-flight between the GPUs processing cores and memory

GeForce2 Ultra GPU, NVIDIA, 2000

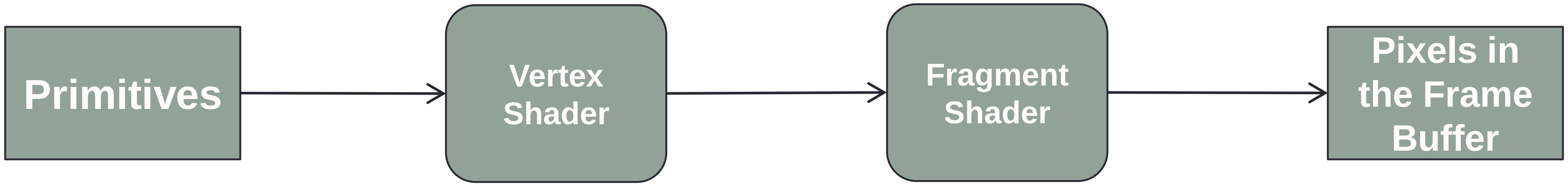
- Software
  - support for pipelines in OpenGL and Direct3D libraries
  - option for pipeline stages to be handled by dedicated (on-chip) hardware
    - not directly supported by OpenCL or CUDA

The OpenGL pipeline

- The OpenGL programming model is based on the pipeline pattern.
- This allows OpenGL compilers/drivers to make optimal use of specific pieces of hardware on the GPU or graphics card.
  - the underlying implementation is based on a lower-level pipeline
- As long as the programmer works with the pipeline model, the details of the hardware implementation can be thoroughly hidden from the programmer.

The OpenGL pipeline continued

- The original version had a complex pipeline.
  - alternative paths determined by special modes of operation
- More recent versions have a simpler pipeline with programmable stages.
  - closer to GPGPU programming with CUDA or OpenCL



- Please note, this is just an overview of OpenGL intended to illustrate the relevance of the pipeline pattern.
  - http://www.cs.utexas.edu/~fussell/courses/cs384g-spring2016/lectures/lectures.html

Pipelines – Conclusion

- Pipelines exist at various levels within software and hardware.
- The Pipeline Pattern is particularly useful when the problem can be mapped onto underlying hardware, e.g. GPUs, Vector Processors.
- This pattern is also useful more generally and often used together with other patterns for applications characterised by a regular flow of data.
- A generalisation of Pipeline termed workflow (or dataflow) is becoming more and more relevant to large, distributed scientific workflows.

Event Based Coordination – Problem

- Event Based Coordination is a Parallel Algorithm Strategy.

- 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 and ordering constraints.

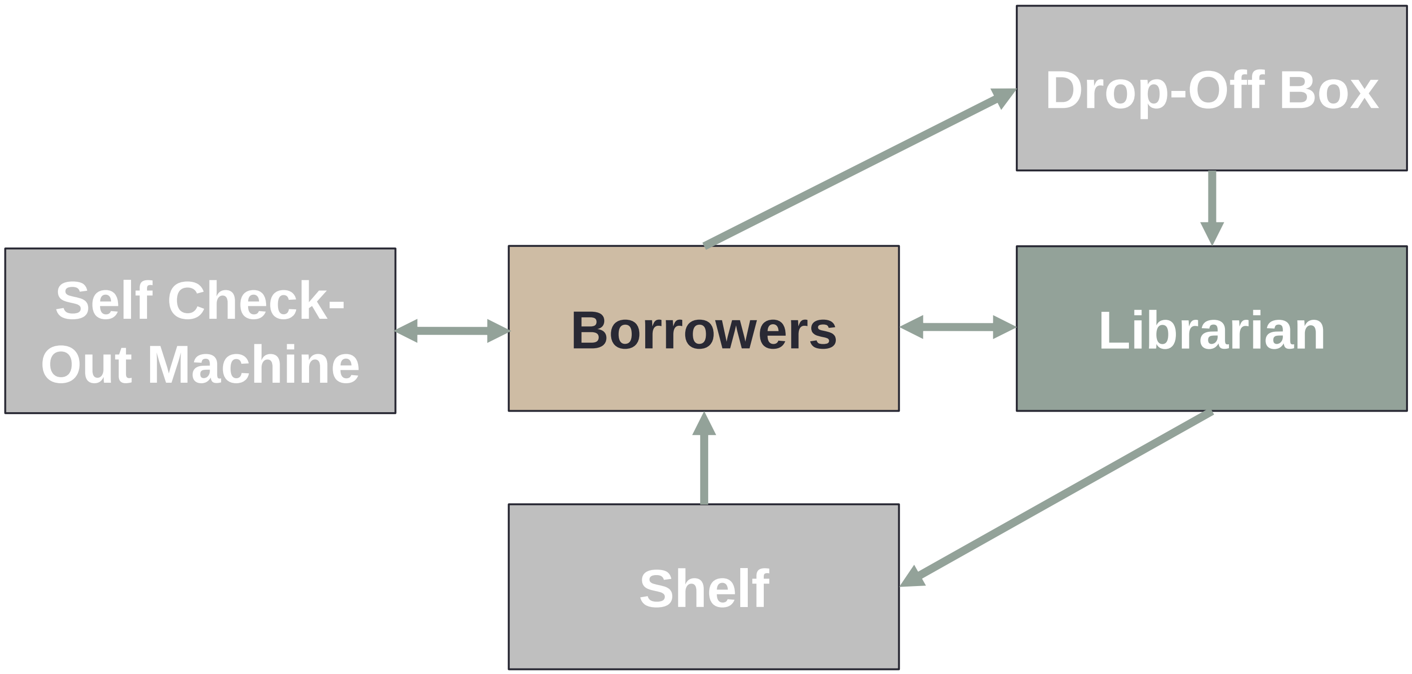
- How can the tasks and their interactions be arranged so that they can run concurrently?

Event-Based Coordination – Context

- Semi-independent entities interacting in an irregular fashion.
- Unlike the Pipeline Pattern there is...
  - no restriction to a linear flow of data
  - any given UE might communicate with any other UE
  - interactions take place at irregular (and unpredictable) intervals
- Related to discrete-event simulation.
  - Simulations of real-world processes, in which real-world entities are modeled by objects that interact through events
- Sometimes desirable to compose existing (possibly serial) program components into a parallel program without changing the components.

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

- This pattern is not just for simulations; it can be applied to real-time applications.
  - monitoring and 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 overs network
    - Google Docs, more complex then you might think!

Event-Based Co-ordination – Forces

- A good solution should...
  - make it easy to express ordering constraints
    - constraints might be numerous and irregular and arise dynamically
  - expose as much parallelism as possible by allowing as many concurrent activities as possible

- Any solution should permit the expression of constraints in ways common to other patterns such as...
  - sequential composition
  - shared variables representing state

Event-Based Co-ordination – Solution

- Events are 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
- A solution consists of...
  - 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 Facade pattern (GoF)

Representing Event Flow

- Allow communication and computation to overlap and to avoid serialisation where possible.
  - events need to be communicated asynchronously
- In a message-passing environment, an 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.

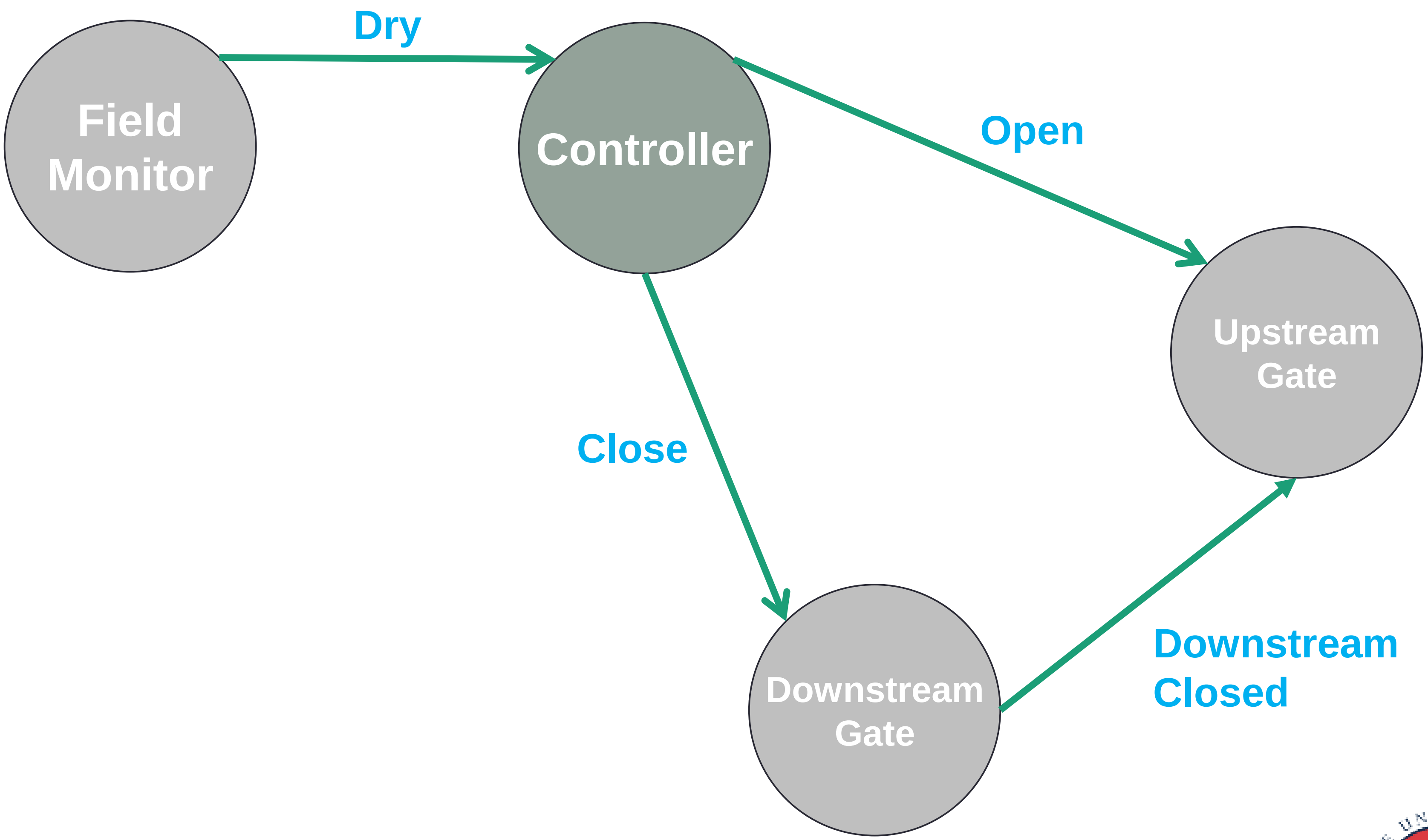
- Other communication abstractions can also be used.
  - Tuple Spaces, JavaSpaces, TSpaces

Enforcing Event Ordering

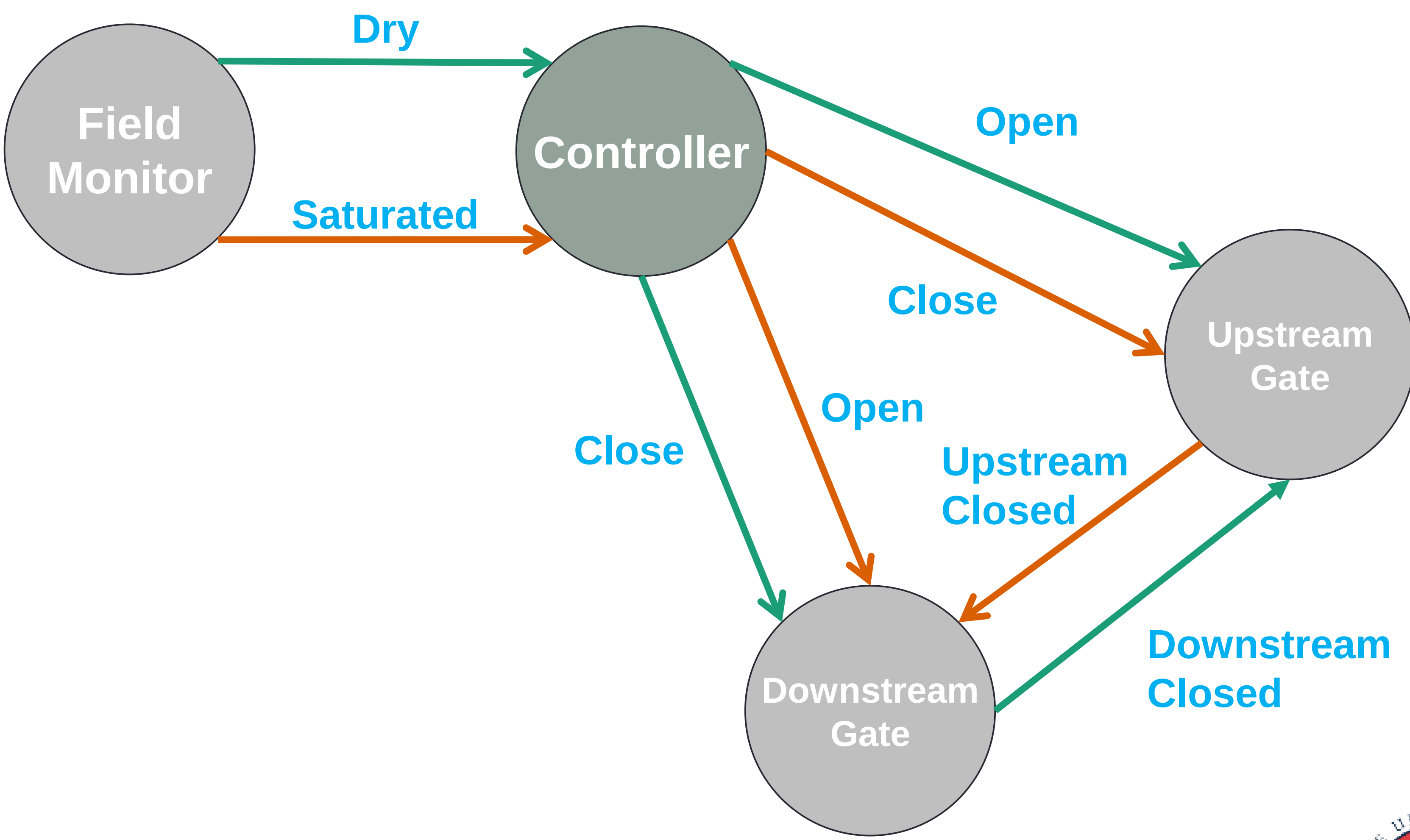
- This is probably the hardest step in applying this pattern.
- Enforcement might require a task to process events in a different order to that received.
  - note the word *received* not *sent*: MPI's guaranteed P2P message ordering is not necessarily enough to protect you here!
  - events are often 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.

Event Ordering – a sluice gate



Event Ordering – a sluice gate



Event Ordering

- Some mechanisms that can help with event ordering constraints are *global clocks* and *synchronisation events*.
- Before you spend time trying to enforce event ordering...
  - check if event path is linear
  - check if application cares whether or not events are out of order

- There are two approaches if ordering does matter.
  - optimistic: proceed and deal with problems later
    - by initiating rollback for example
  - pessimistic: wait for a synchronisation event or similar
    - ensures ordering constraint is met but creates overhead

Avoiding Deadlocks

- A common problem with this pattern.

- You can get the normal message passing deadlocks, but with event-driven simulation you can also have application-level deadlocks.
  - deadlocks caused by implementation error or by “model error”

- Deadlocks can arise from being overly pessimistic.
  - possible to use runtime deadlock detection
    - often inefficient as a general solution
    - *local* timeouts can work well in place of full deadlock detection

Scheduling and Processor Allocation

- Most straightforward approach is one task (and one UE) per processing element (PE)
  - allows all tasks to execute concurrently
- Also possible to have several tasks and therefore several UEs per PE.
  - suboptimal 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.
  - therefore, 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 split them perhaps) to improve efficiency.
- With a shared-memory environment, care must be taken that shared queues are implemented efficiently, as these can easily become bottlenecks.

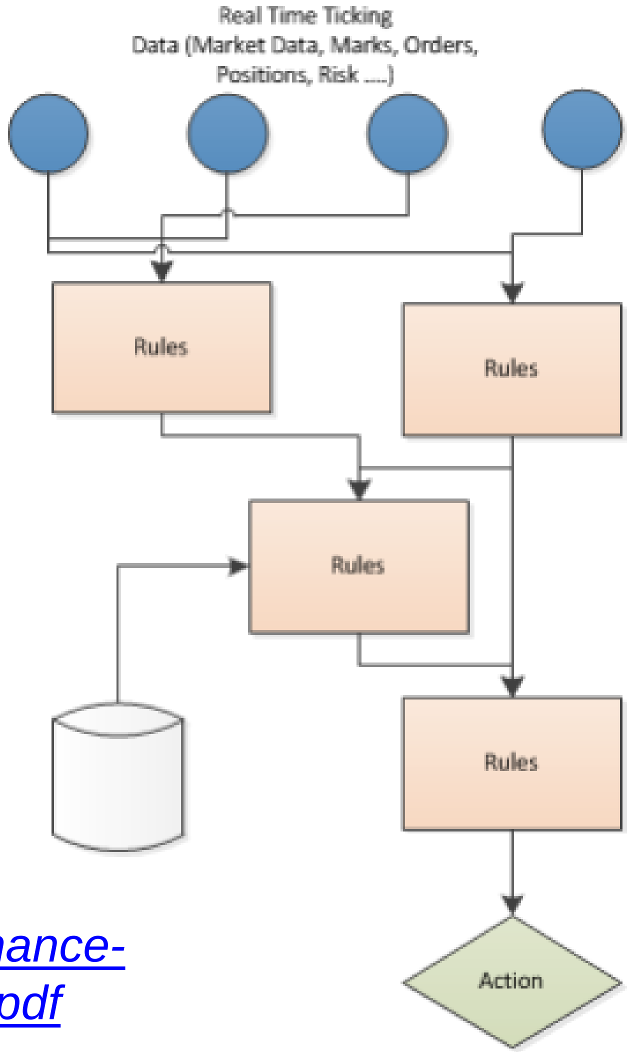
Comments

- Sometimes you can get increased parallelism if you can work with partial answers.
  - may only need a certain number of events to compute a result
  - UEs in the system can work on part of the problem asynchronously and return "*best current guess*" in response to a received event
  - the "*best guess*" can be subsequently refined if required

- This pattern is closely related to the Actor pattern.

UBS Financial Information Exchange

- Real time market data for quotes, orders & executions
  - peak bandwidth of 16 million items per sec
  - low latency: events processed in msec
- Stages run concurrently...
  - in a separate thread on custom hardware
  - "*compute in the data plane*"
- Low level optimisation...
  - at networking, OS and runtime level
  - using FPGA based hardware



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

Event-Based Coordination: Summary

- Designed for problems characterised by irregular flow of data.

- Maps real-world entities to tasks.
- Models real-world interactions with events.
- The hardest part to get right is often the event ordering.
  - communication is not instantaneous, whereas the real-world interactions you're modelling often are synchronous.
- Model details not necessarily dependent on parallelisation strategy of code.

- Pattern can be applied to existing code components.