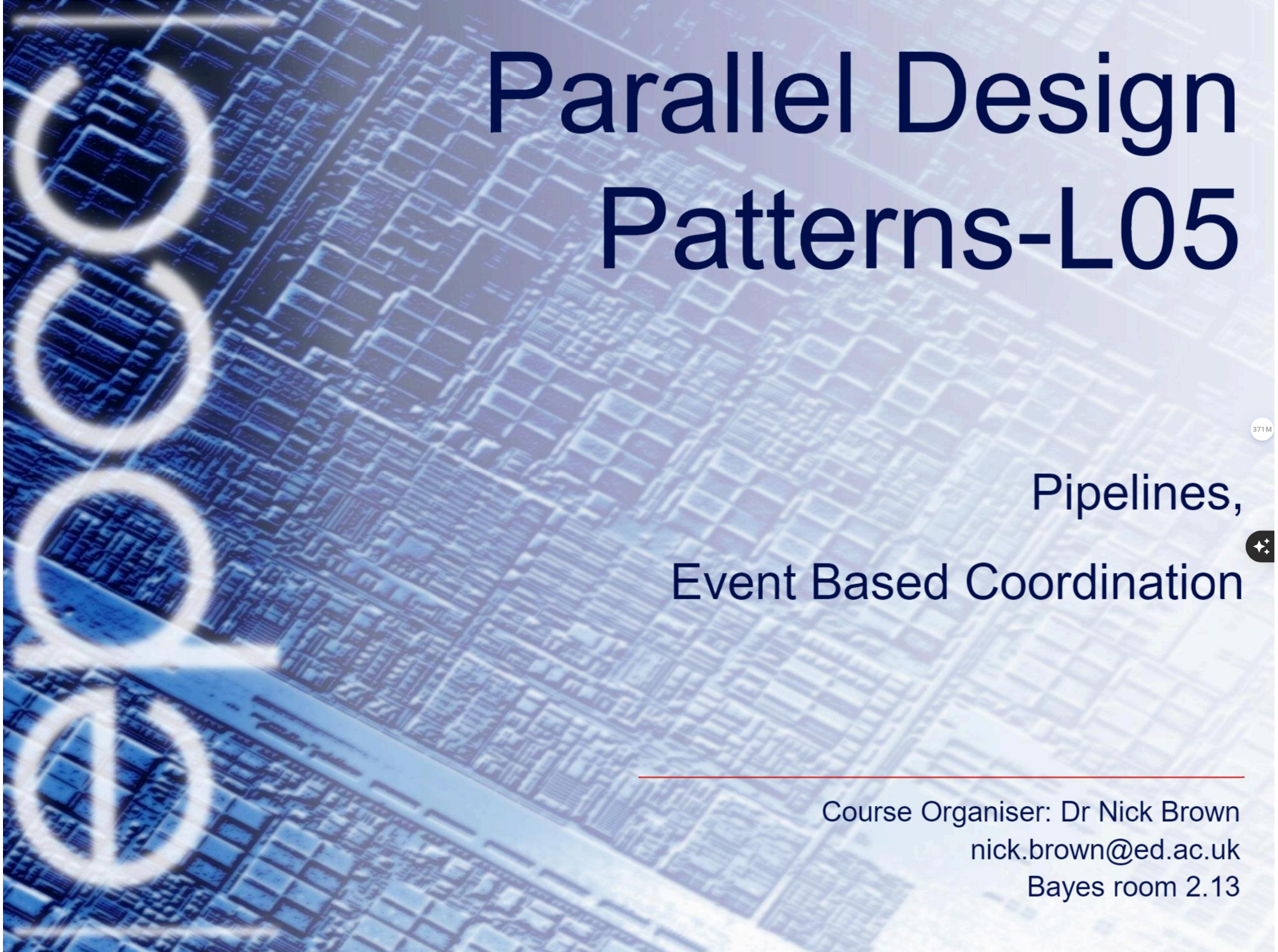


Parallel Design Patterns-L05

Pipelines,

Event Based Coordination

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

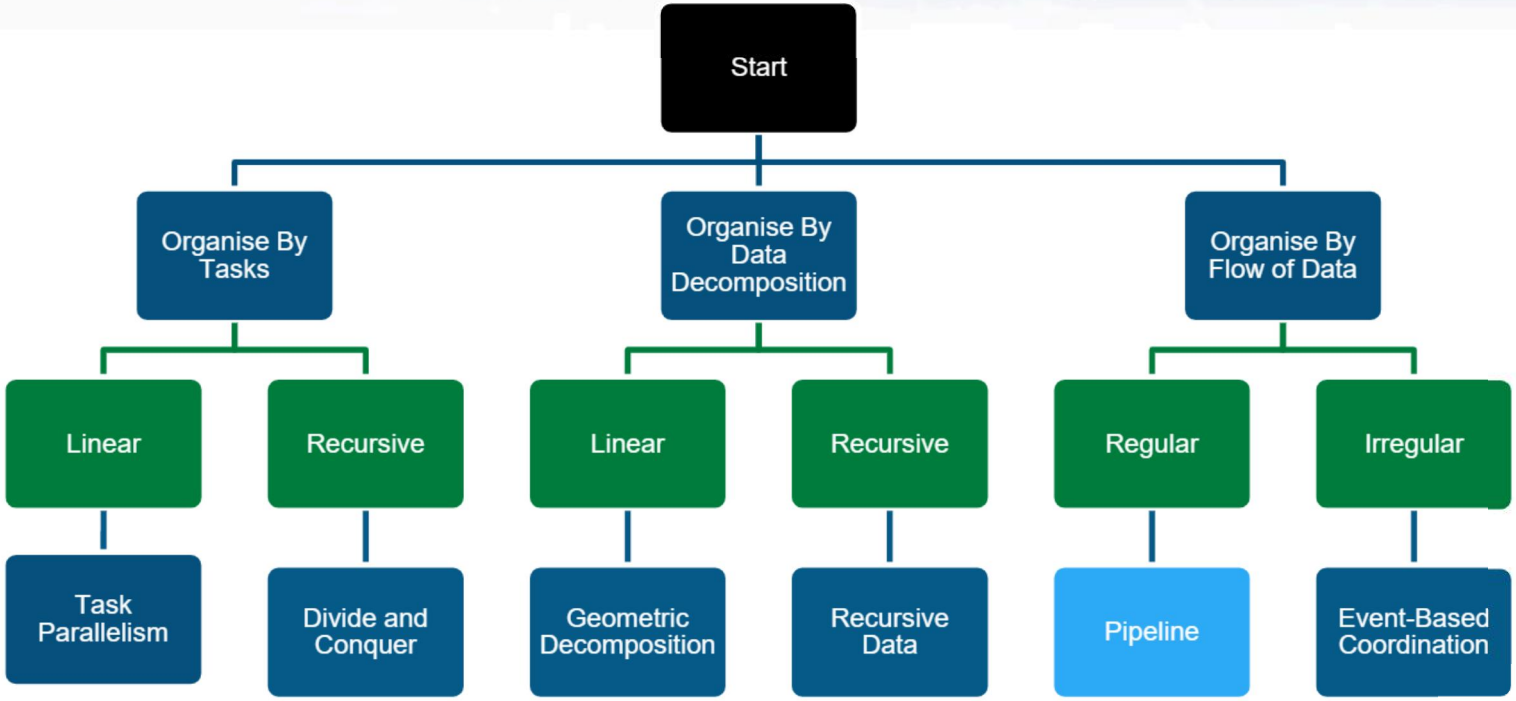
并行设计

图案-L05

管道，

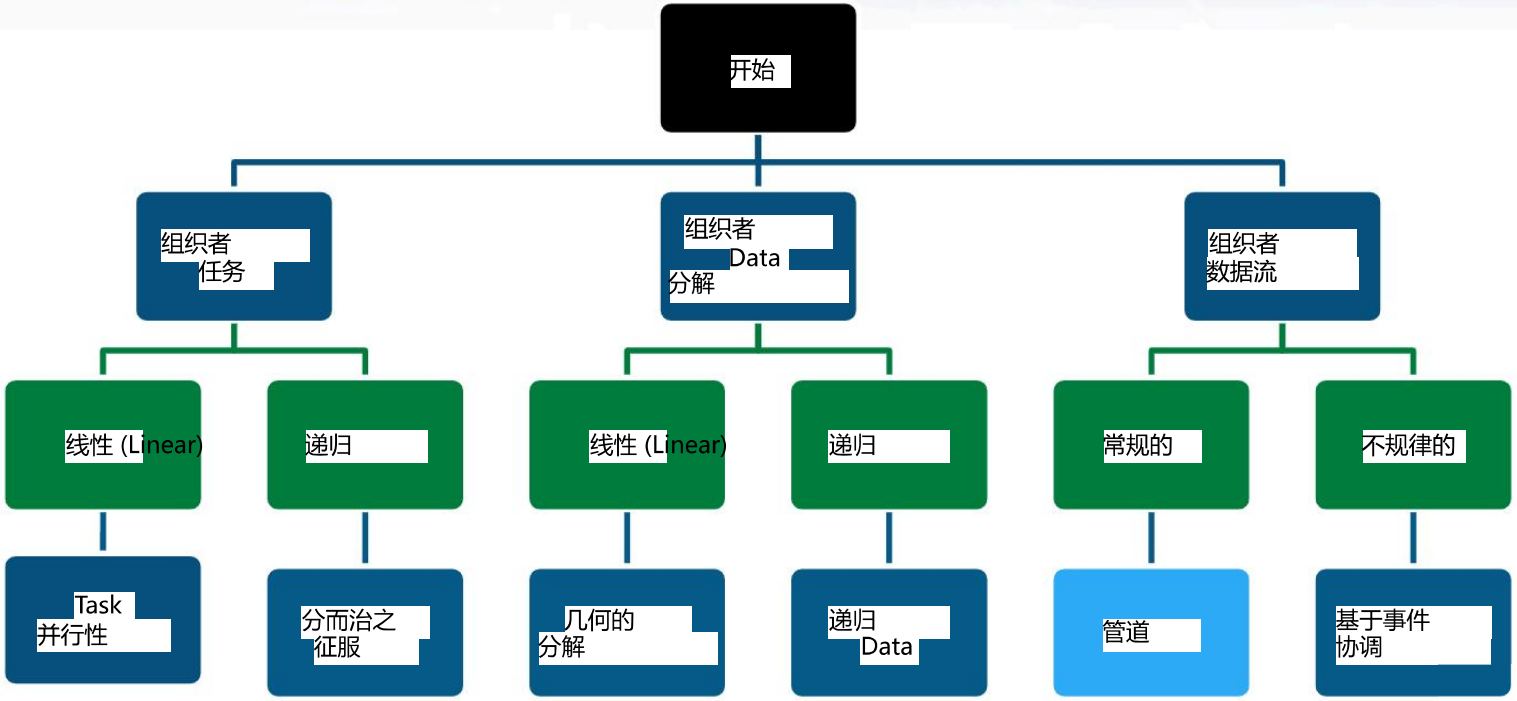
基于事件的协调

课程组织者：Nick Brown 博士
nick.brown@ed.ac.uk
贝叶斯室 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?



管道模式：

一个问题涉及对多条数据进行依次操作。整体计算可以看作是数据流经一系列阶段，并在每一阶段进行操作。如何利用潜在的并行性？

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

管道——背景

- 流水线是一个很好的类比
  - 我们拥有的不是部分组装的汽车，而是数据 – 我们拥有的不是工人或机器，而是用户
- 管道有多种粒度级别
  - CPU 中的指令流水线 – 矢量处理（循环级流水线） – 算法级流水线 – 信号处理 – UNIX 中的 Shell 程序
- 管道模式利用涉及具有直接排序约束的任务的问题
  - 按顺序对每个数据元素进行多项操作 – 每块数据都有顺序约束，但关键的是，操作可以在不同的数据元素上同时运行

Pipeline – The Forces

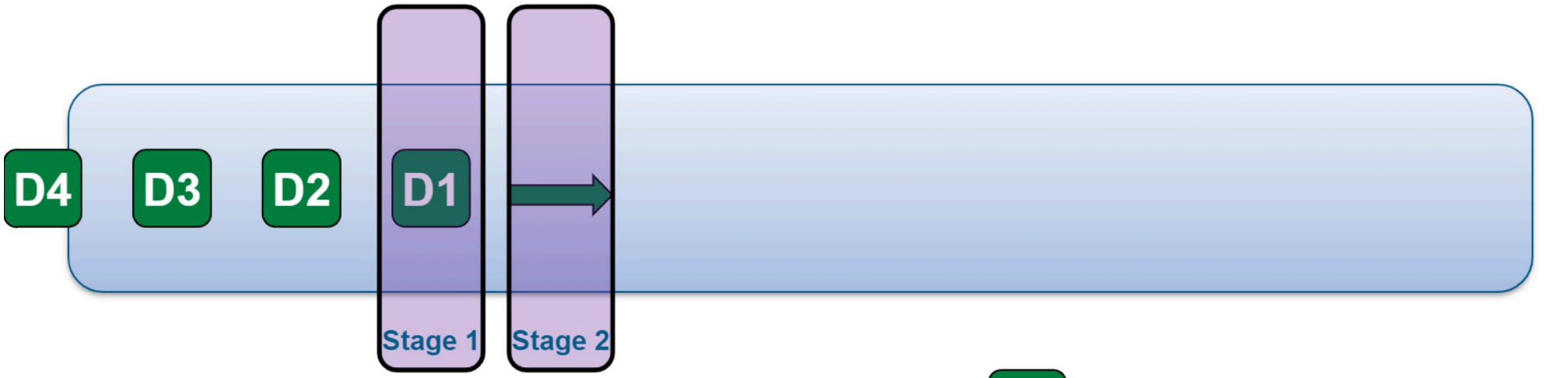
- A good solution should make it easy to express ordering constraints
  - These are simple and regular
  - Lend themselves to being expressed as data flowing through a pipe
- Target platform should be borne in mind
  - Sometimes contains special hardware to perform some of this
- In some applications,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

管道——力量

- 一个好的解决方案应该能够轻松表达排序约束
  - 这些都是简单而有规律的 – 可以表示为流经管道的数据
- 应牢记目标平台
  - 有时包含特殊硬件来执行其中一些操作
- 在某些应用中，未来可能会对管道进行修改（或重新排序）

Pipeline – The Solution

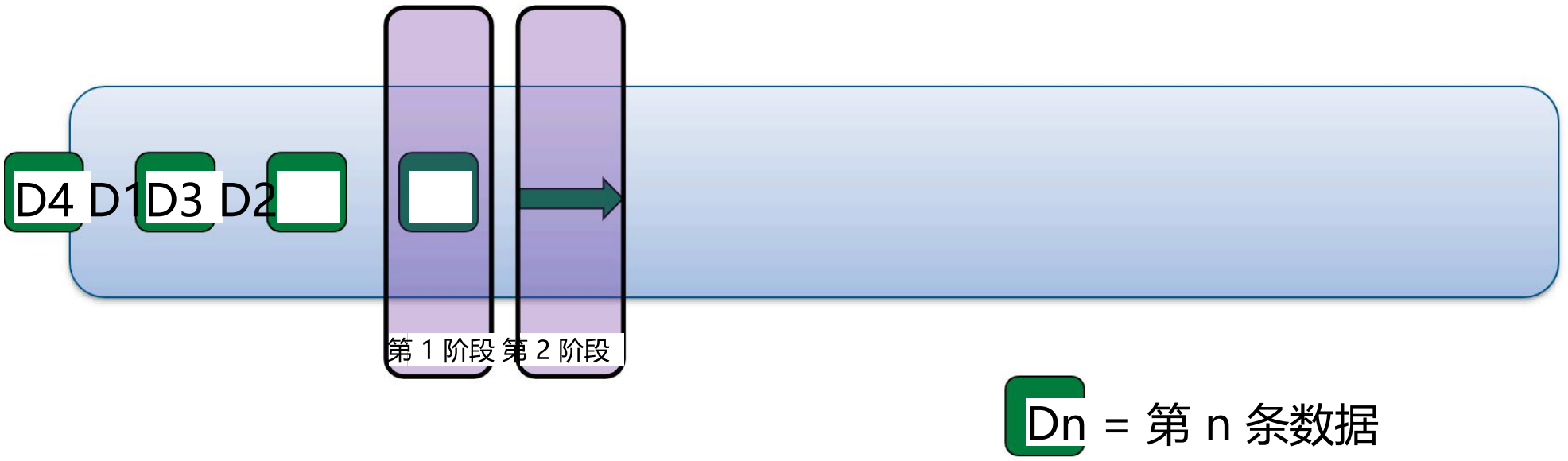
- Idea is captured by the assembly line analogy
- Assign each computation stage to a different UE and provide a mechanism to so that each stage of the pipeline can send data elements to the next stage



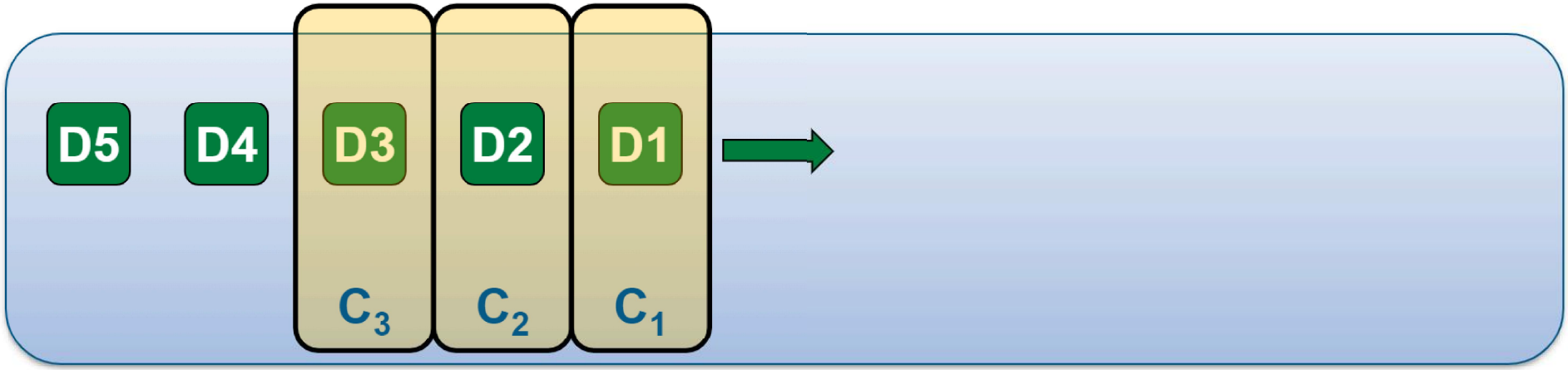
D*n* **= the *n*th piece of data**

管道——解决方案

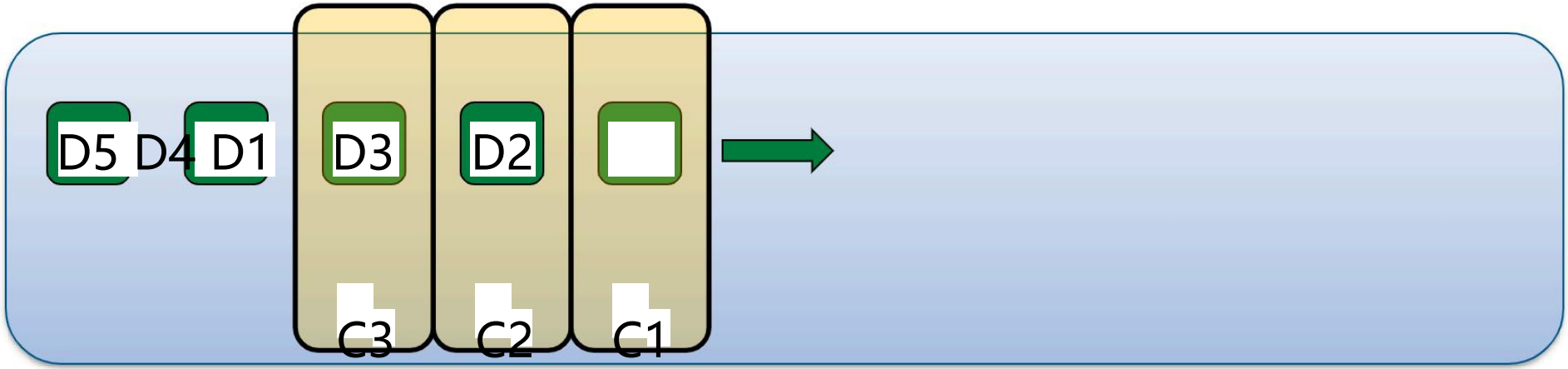
- 这个想法可以用流水线类比来表达
- 将每个计算阶段分配给不同的 UE，并提供一种机制，以便管道的每个阶段都可以将数据元素发送到下一个阶段



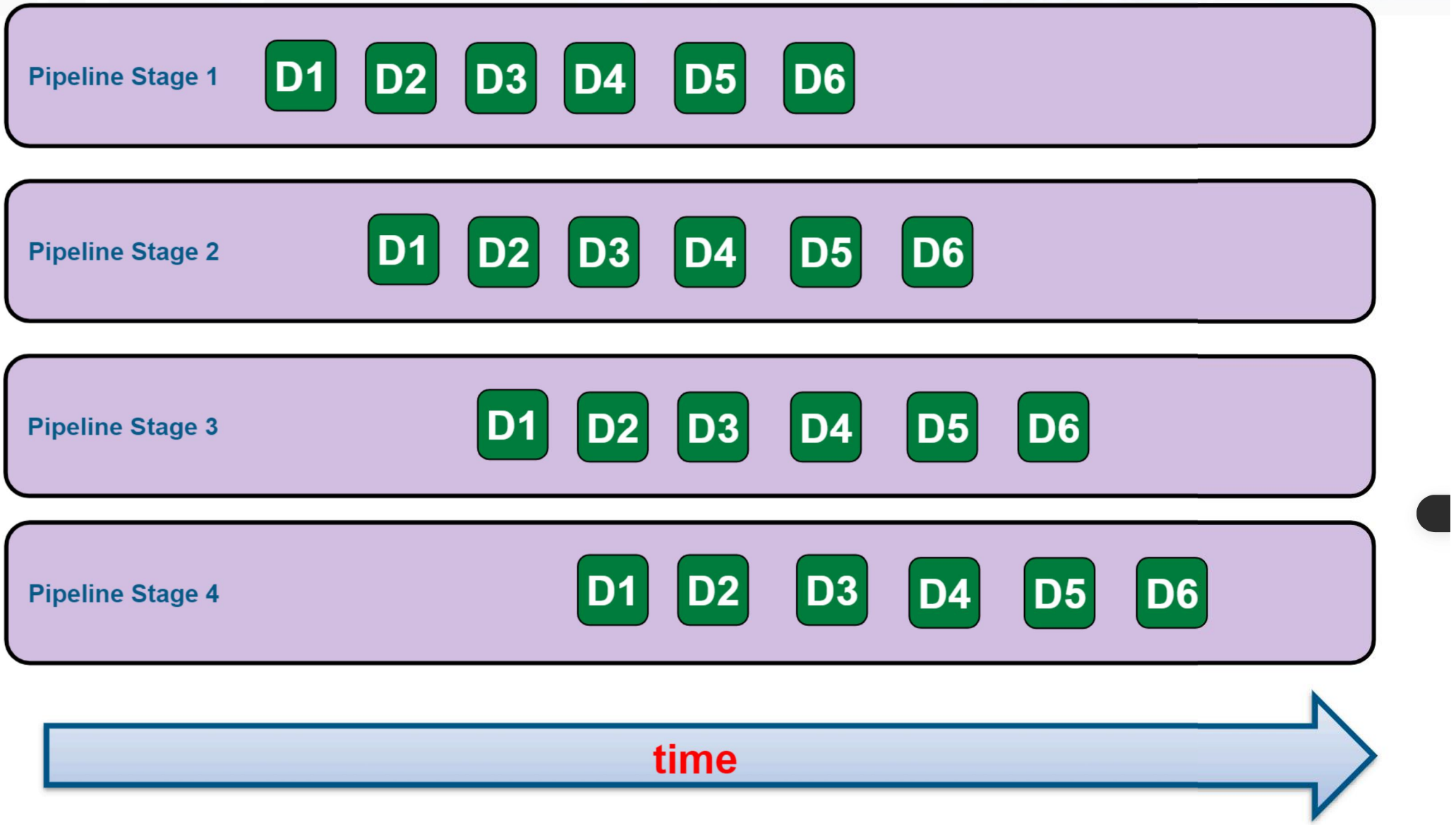
Snapshot of pipeline at $t = 3$



t = 3 时的管道快照

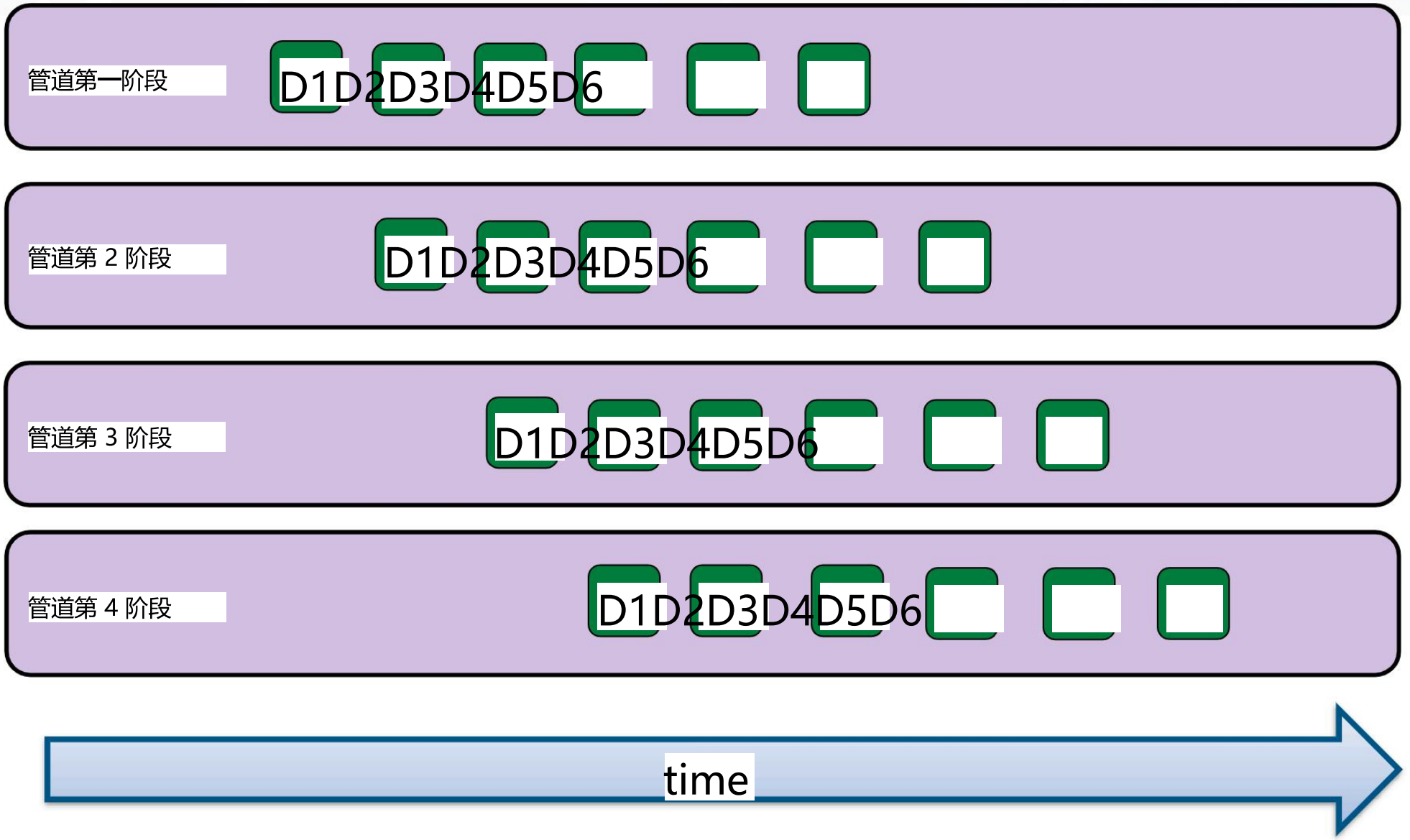


Pipeline example



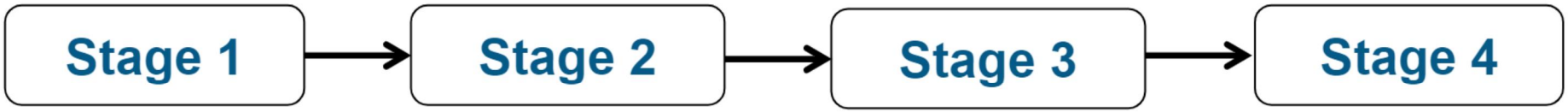
At time=4 the pipeline is filled, starts to drain at time=7

管道示例

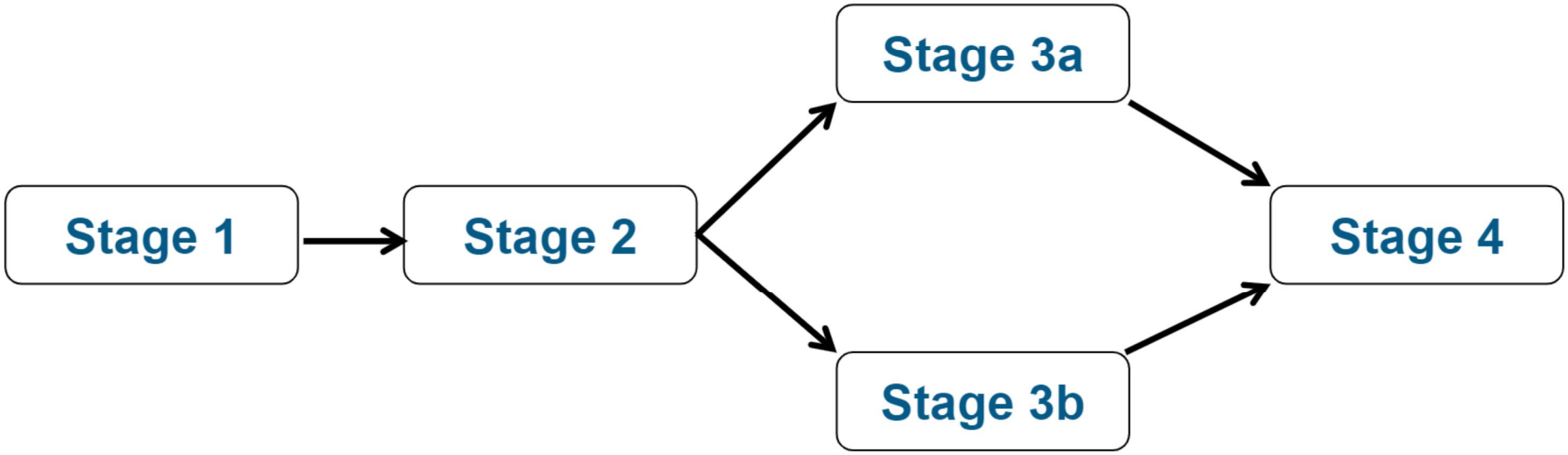


在时间 =4 时管道已充满，在时间 =7 时开始排水

Pipeline architectures



Linear pipeline

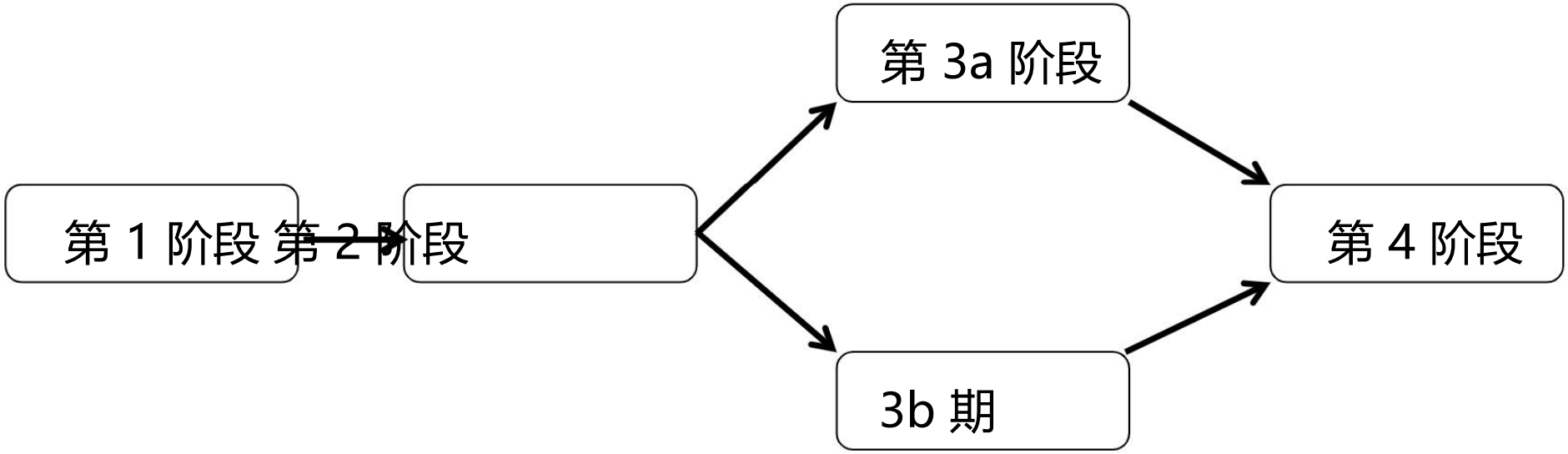


Non linear pipeline

- 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

管道架构

线性管道



非线性管道

- 吞吐量（带宽）是单位时间内可以处理的数据项数量（一旦管道已满）。
- 延迟是处理单个数据元素所需的时间

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

定义管道的阶段

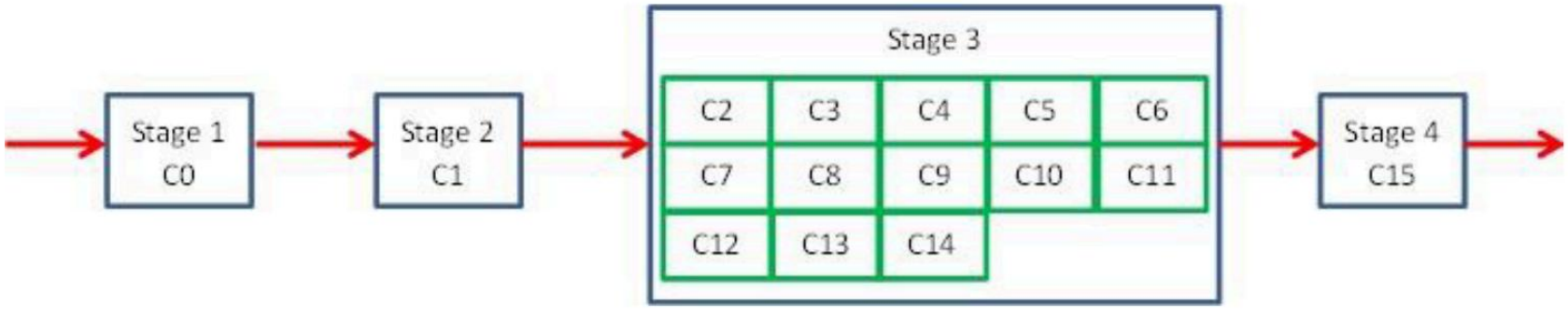
- 通常每个流水线阶段对应一个任务
- 管道阶段关闭时
  - 它计算出它已经完成了所有任务（如果已知数量）
  - 通过管道接收关闭的“哨兵”（毒丸）
- 并发性受阶段数限制
  - 但数据必须在各阶段之间传输
- 如果所有阶段的计算强度都相似，则模式效果最佳

- 如果填充/排出管道的时间与运行时间相比较小，则该模式效果最佳

或者，如果延迟相对于带宽较小，则取决于管道深度、数据元素数量

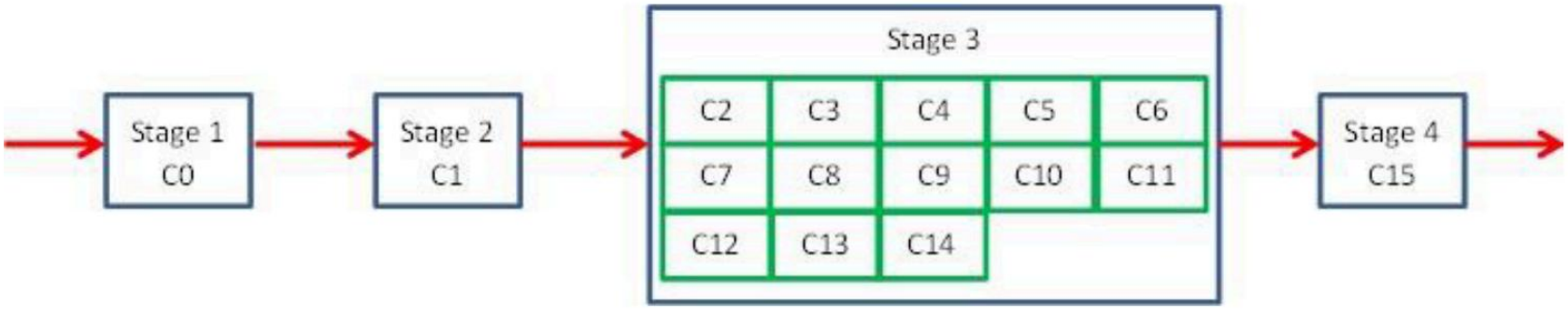
Structuring the computation

- 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*



构建计算

- 需要定义整体计算
- 管道常用
  - SPMD 模式，使用 UE 的标识符来区分案例陈述中的选项
- 模式可以与其他算法策略相结合，帮助平衡各阶段之间的负载
  - 例如，一个流水线阶段可以与任务并行化



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

表示数据流

- 由语言/架构中可用的支持结构驱动

- 使用 MPI：将元素之间的数据流映射到消息
  - 点对点通信 – 管道中每个阶段对应一个进程

- 对于共享内存：使用共享队列模式
  - 我们将在后面的课程中看到

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

管道代码草图

- 每个管道阶段将具有以下结构：

初始化
当（未完成）
{ 阻塞 从上一阶段接收数据 处理数据 将处理后的数据发送到下一阶段 } 将终止标记发送到下一阶段 完成

- 数据的发送可以是非阻塞的，也可以是缓冲的
- 您的终止哨兵可以是一条空消息，您可以检查收到的数据元素数量

Handling errors

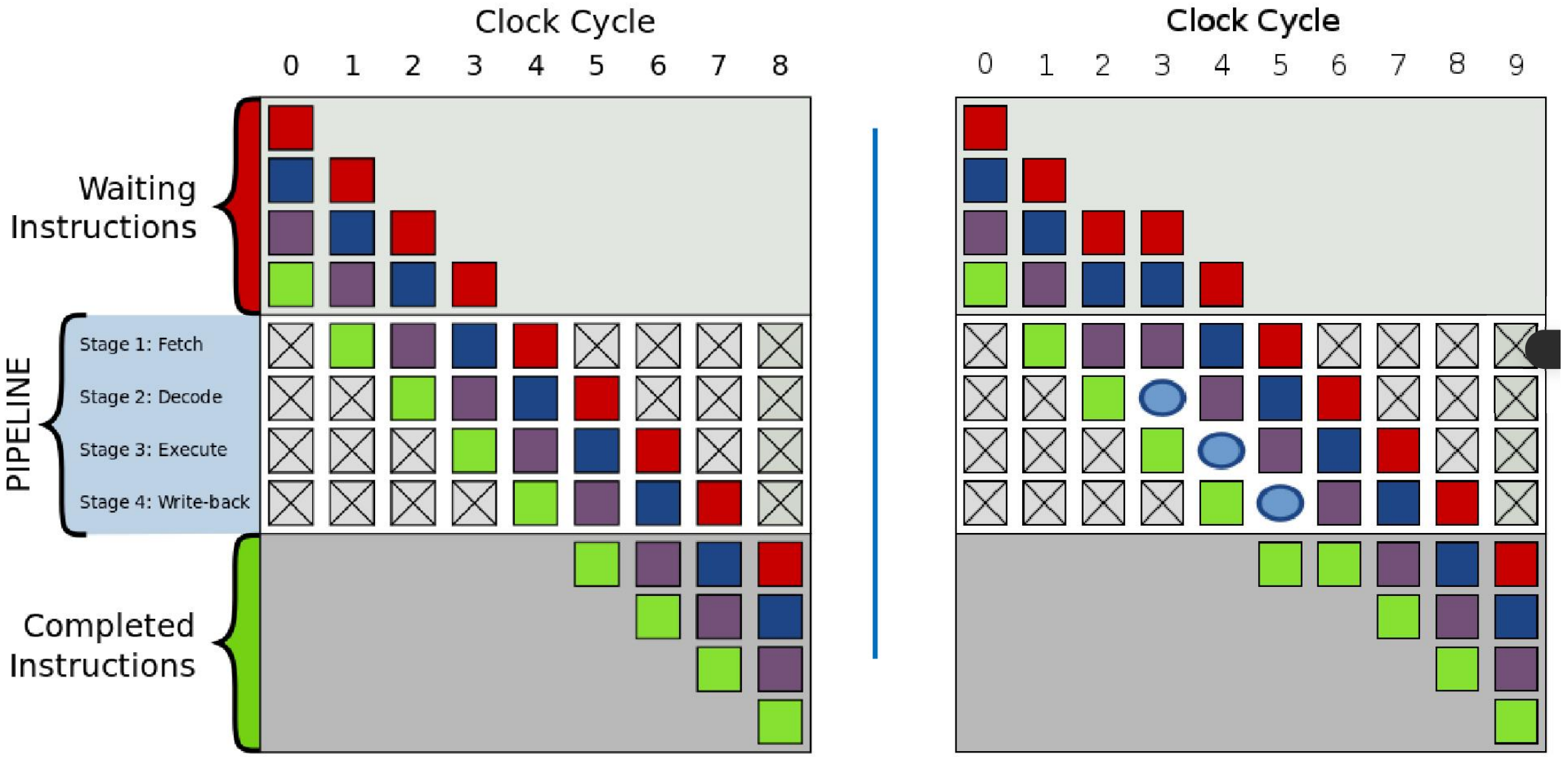
- Potential for “a spanner in the works”
- If there’s an error in one part of the pipeline, it has potential to break the whole flow
  - Do I care? Sometimes no, but sometimes yes – especially if you need a 1:1 correspondence between input tokens and output tokens
- Common solution is to have a separate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

处理错误

- 可能造成“阻碍”
- 如果管道的某个部分出现错误，则可能会打破整个流程
  - 我关心吗？有时不关心，但有时关心——特别是当你需要输入标记和输出标记之间 1:1 的对应关系时
- 常见的解决方案是有一个单独的“错误处理”任务（或任务）与管道元素可以进行通信
  - 保持管道畅通
    - 传递一个“错误”标记，通常称为气泡 – 在每个后续阶段充当“noop”

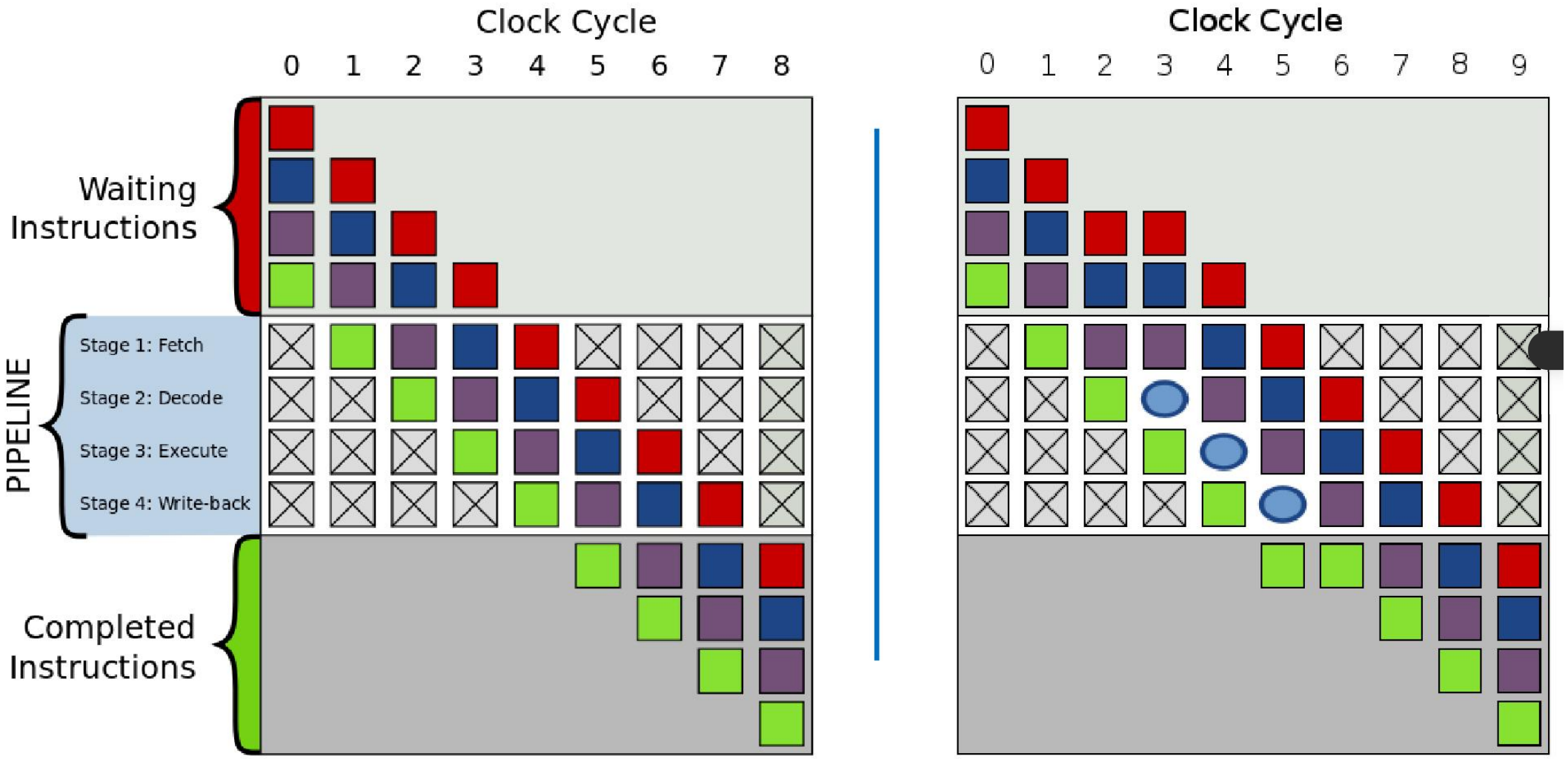
Instruction Pipelines

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



指令流水线

- 4 个活动/指令：获取、解码、执行、写回



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

处理器分配和任务调度

- 最简单的方法：每个管道阶段一个用户设备
  - 如果每个管道阶段的工作量大致相同，则负载平衡良好

- 如果 UE 数量少于流水线级数：
  - 超额订阅：将多个阶段分配给同一个 UE。理想情况下，这些阶段不共享资源。通过将相邻阶段分配给单个 UE 可以减少通信开销，或者 – 将阶段合并为一个更大的阶段

- 如果 UE 数量多于流水线级数：
  - 并行化一个阶段（管道内的任务并行），或 – 运行多个管道（任务并行内的管道），只要数据之间没有时间约束

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

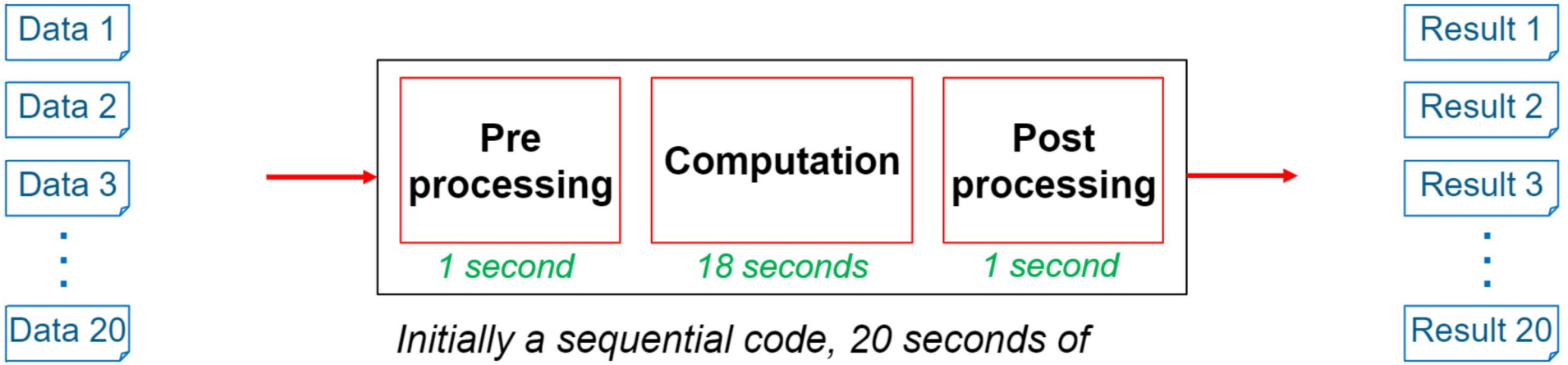
UNIX 指令流水线

```
cat 数据文件 | grep  “energy”  | awk'{print $2, $3}'
```

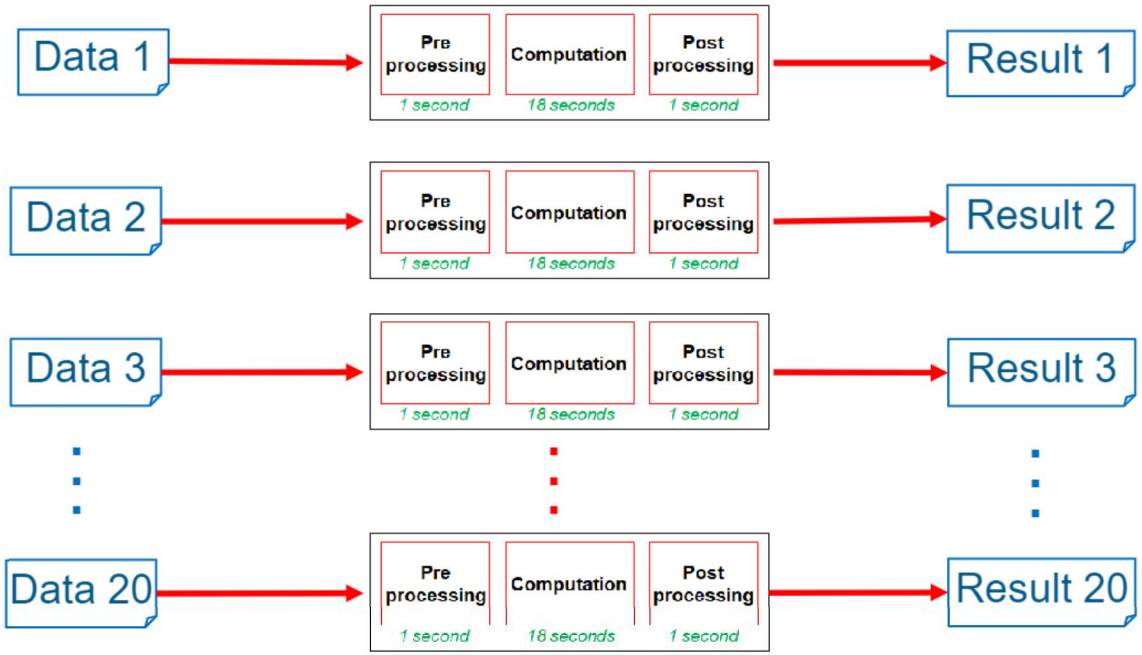
- 通过启动三个进程并使用共享队列实现的缓冲区来实现
- 进程通过其 stdin 和 stdout 流连接
- 作为匿名管道实现（进程完成后立即断开）
- UNIX 还提供可以在程序调用之间持续存在的命名管道

  – 看起来像文件 – 使用 mkfifo 命令创建

Computation example

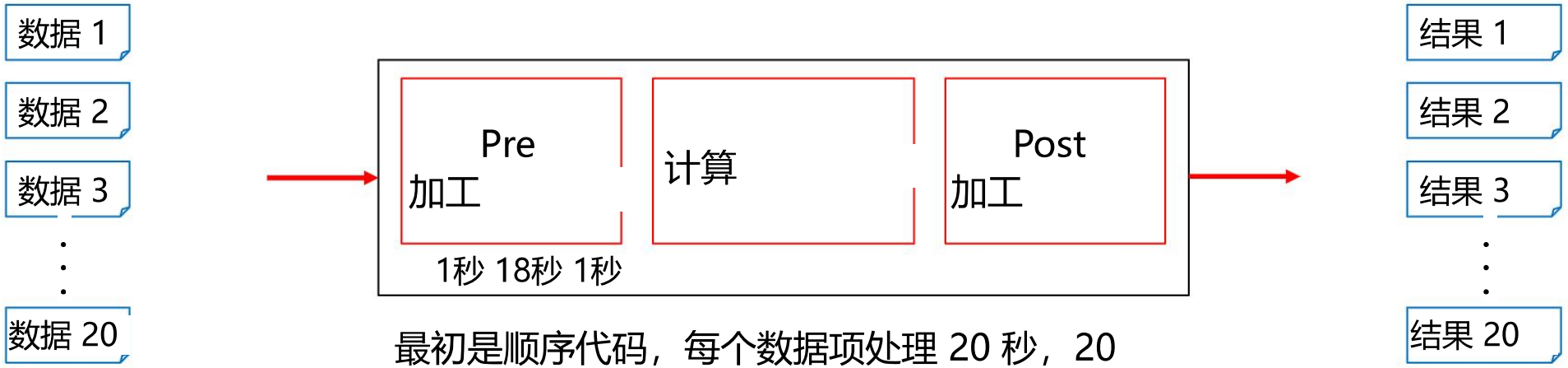


*Initially a sequential code, 20 seconds of processing per data item, 20 items of data: 20*20 = 400 seconds runtime*

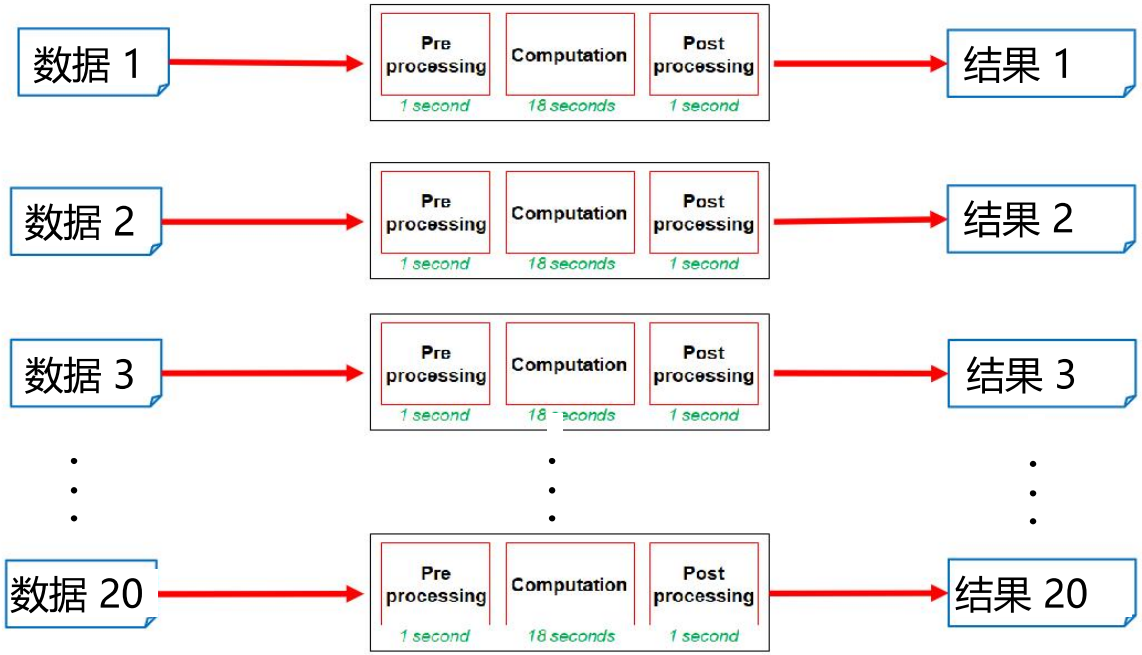


- *Parallelised this by splitting up into 20 separate tasks, one task per element of data*
  - *Run over 20 UEs*
- *Overall runtime is now 20 secs*
- *But no results are generated until 20 seconds, at which point all results are available*

计算示例



最初是顺序代码，每个数据项处理 20 秒，20 个数据项：20*20 = 400 秒运行时间

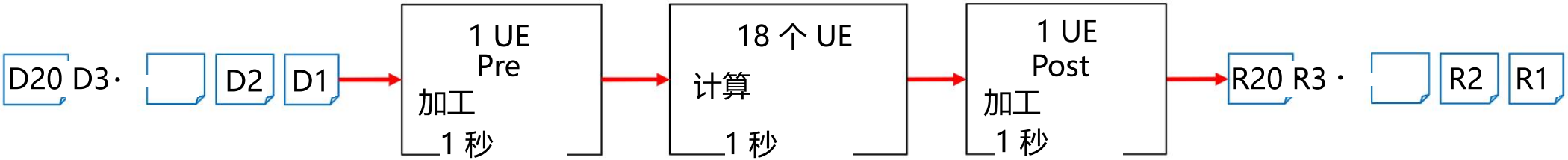


- 将其拆分成 20 个独立任务，每个数据元素一个任务，从而实现并行化
  - 运行超过 20 个 UE
- 总运行时间为 20 秒
- 但直到 20 秒才会生成结果，此时所有结果都可用

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

计算示例



- 使用并行模式（例如几何分解）将 18 个 UE 分配给计算阶段，将 1 个 UE 分配给其他两个阶段。

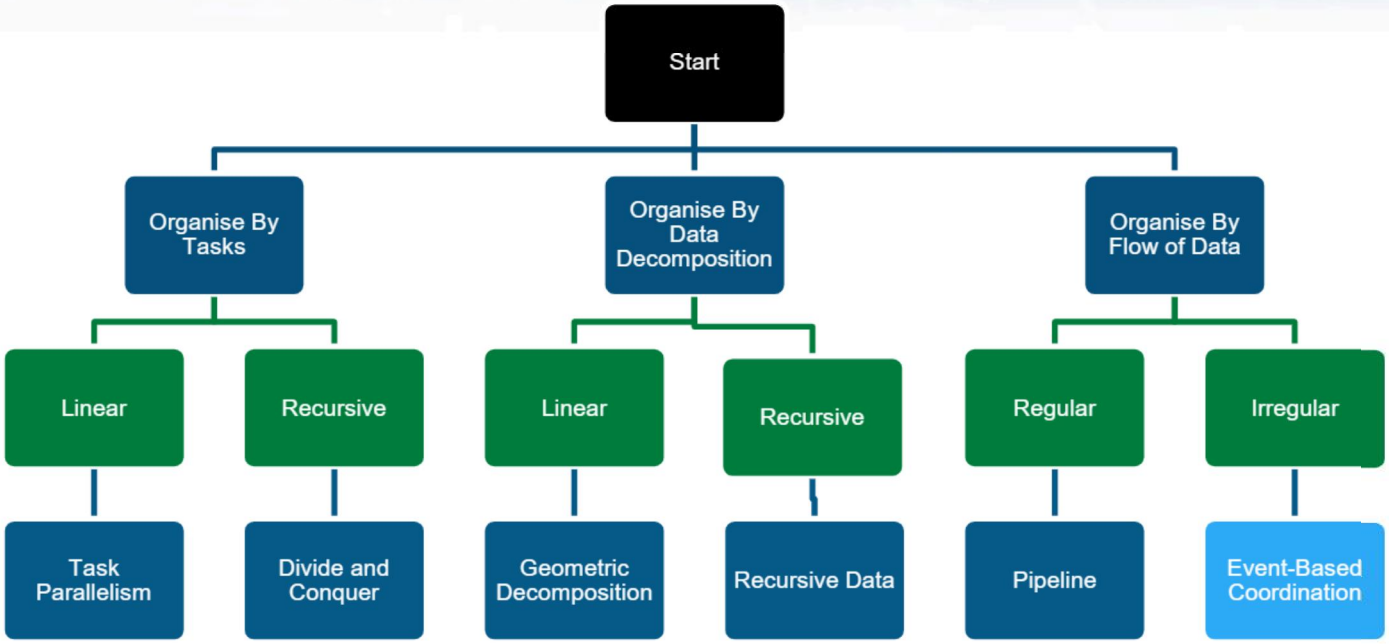
- 因此每个阶段现在需要 1 秒 = 总共 3 秒
- 如果我们不以管道的方式运行
  - 当前一个元素完全处理完毕后，才开始处理数据的后续元素
  - 处理每个元素需要 3 秒（每 3 秒生成一个结果）
  - 3 * 20 = 60 秒总运行时间
- 如果我们以管道形式运行
  - 每个阶段同时运行，处理不同的数据
    - 处理第一个数据元素仍需要 3 秒，但一旦完成并且管道完全填满，则每秒都会生成一个结果
    - 3 + 19 = 22 秒总运行时间
    - 这比对每个数据元素进行单独的任务处理要稍微长一点，但至关重要的是，一旦管道填满，我们每秒都会生成一个结果

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

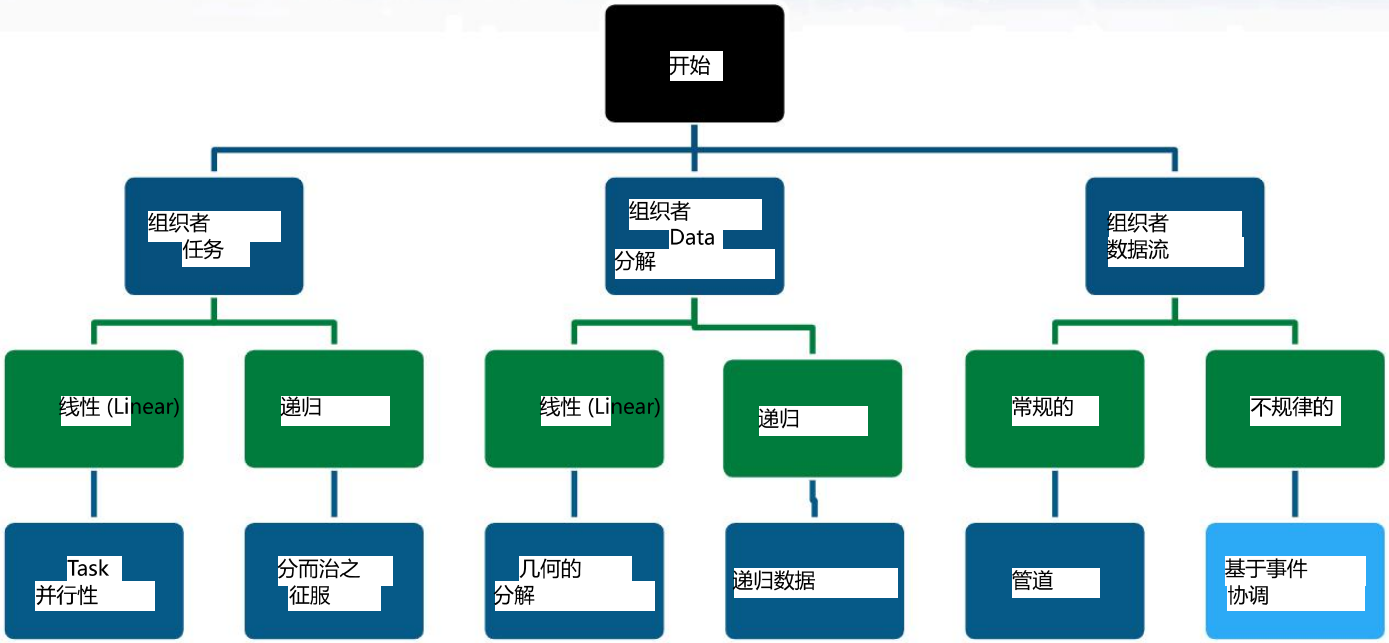
管道：结论

- 管道存在于软件和硬件的不同层次
  - 构成处理器微架构设计的基石
- 该模式也具有更广泛的用途，并且经常与其他模式一起使用，用于以常规数据流为特征的应用程序
  - 例如数据预处理和后处理
- Pipeline（工作流或数据流）的泛化与大型分布式科学工作流越来越相关



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?



基于事件的协调模式：

问题：应用程序可以分解为以不规则方式交互的半独立任务组。交互由数据流决定，这意味着排序约束。如何安排任务及其交互，以便它们可以同时运行？

Event-Based Coordination: Context

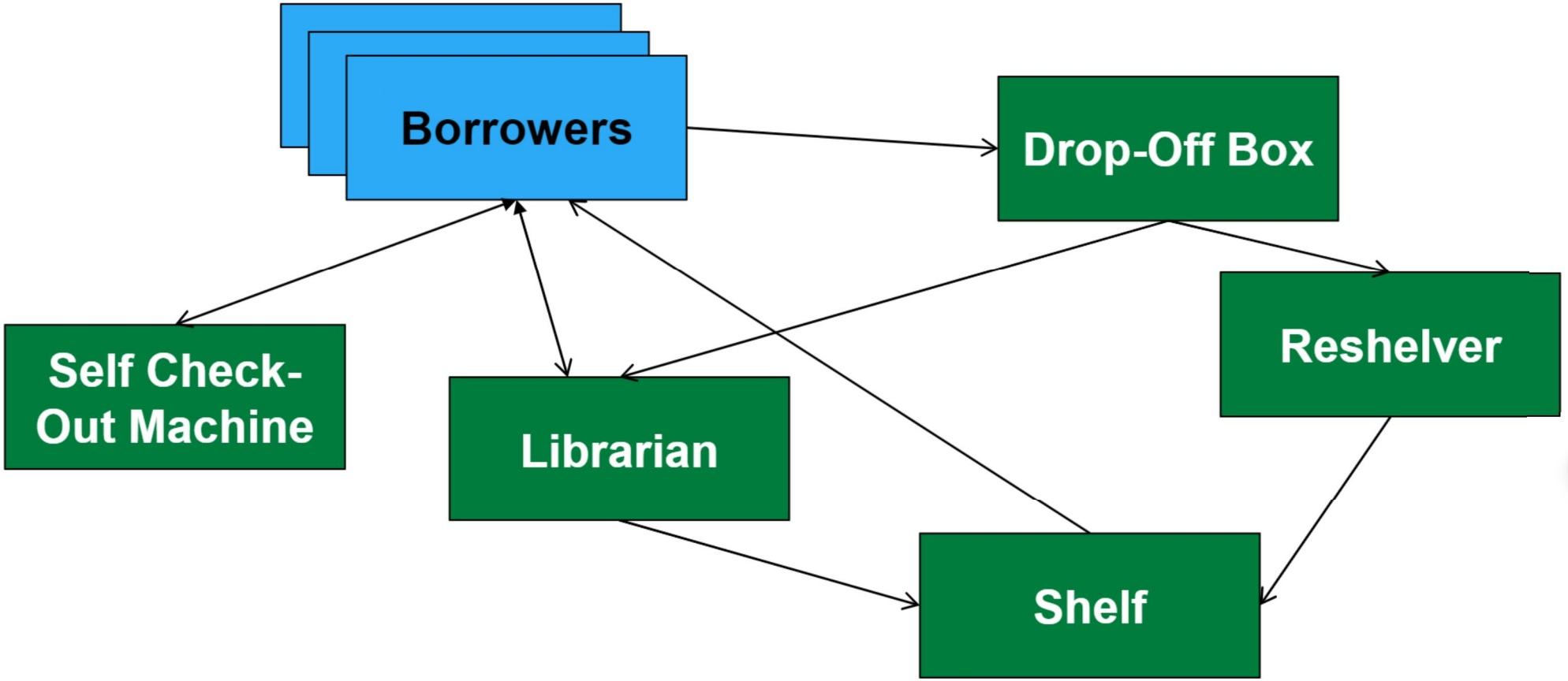
- Semi independent entities which are interacting in an irregular fashion
  - Events sent from one task to another and processed by a *handler*
- In contrast to *Pipeline:*
  - No restriction to a linear flow of data
  - Might communicate with any other task
  - Interaction can take place at irregular (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

基于事件的协调：背景

- 以不规则方式互动的半独立实体
  - 事件从一个任务发送到另一个任务并由处理程序处理
- 与管道相比：
  - 不限制数据的线性流动 – 可以与任何其他任务进行通信 – 交互可以以不规则（且不可预测）的间隔进行 – 不同的事件（数据）可以用非常不同的方式处理
- 与离散事件模拟密切相关
  - 模拟现实世界的过程，其中现实世界的实体由通过事件交互的对象建模

Discrete-Element Simulation

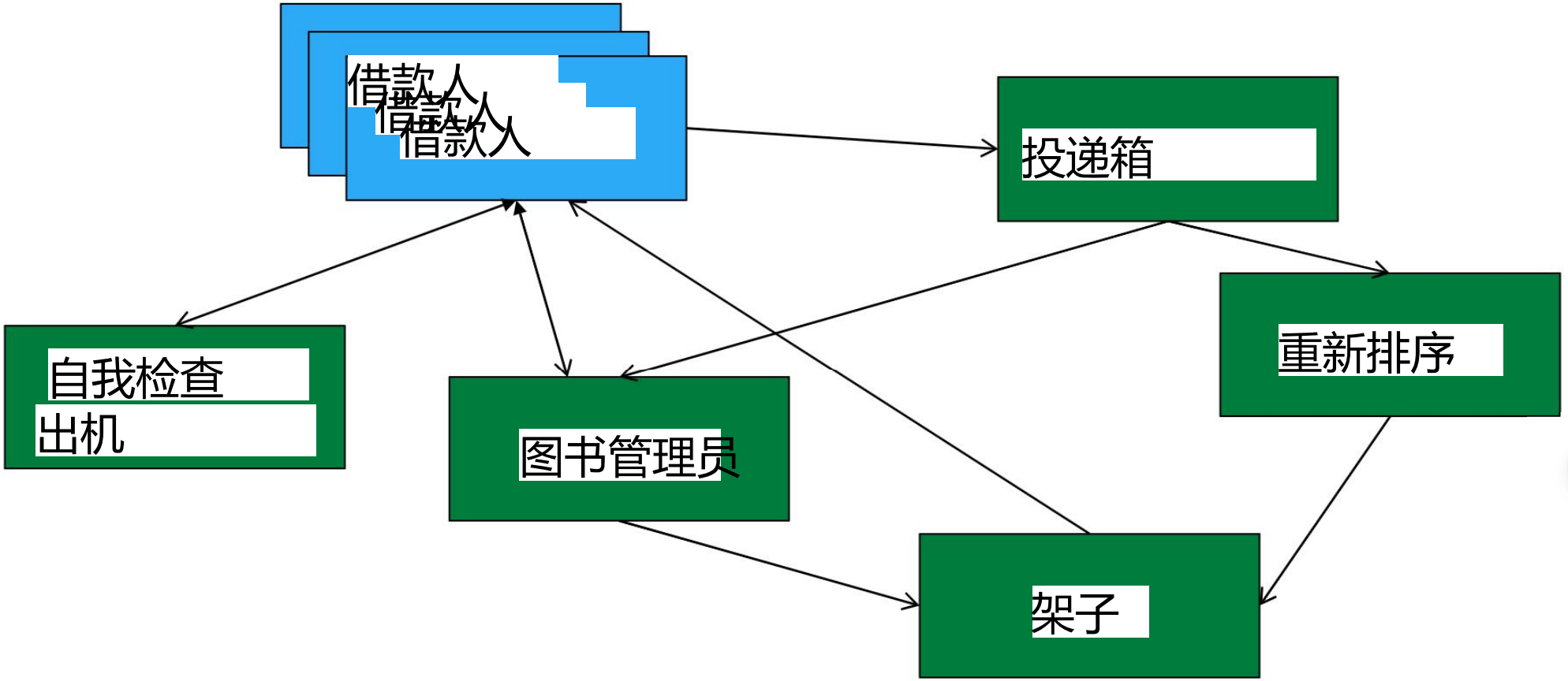
- Example: Modeling the flow of books in a library:



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

离散元模拟

- 示例：对图书馆中的图书流动进行建模：



至关重要的是，每个阶段从根本上来说仍然在响应一些数据（事件）到达它，否则不执行任何操作

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!

事件驱动的应用程序

- 不仅仅用于模拟
- 模式可应用于实时应用程序
  - 例如发电站的监测/控制系统
- 大多数基于 GUI 的应用程序都是事件驱动的
  - 事件来自用户（键盘、鼠标等）和系统 – 不是大规模并行，但可以从并行性中受益
- 分布式应用程序
  - 事件通过网络传递 – 例如 Google Docs – 比您想象的更复杂！

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

基于事件的协调：力量

- 一个好的解决方案应该
  - 使表达排序约束变得容易 – 可能数量众多、不规则、动态出现 – 尽可能多地展示并行性 – 通过允许尽可能多的并发活动
- 该解决方案应该提供一种替代方法，以尝试以其他方式表达约束，例如可能在其他模式中找到的约束，例如
  - 顺序组合 – 表示状态的共享变量

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

基于事件的协调：解决方案

- 事件——从一个任务（源）发送到另一个任务（接收器）
  - 暗示排序约束 – 计算包括处理事件
- 每个现实世界实体一个任务
  - 通常每个任务一个 UE
- 解决方案的要素 – 定义任务 – 表示事件流 – 强制事件排序 – 避免死锁 – 调度处理器分配 – 高效的事件通信

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)

定义任务

- 每个任务具有以下结构：

```
初始化
当（未完成）
{
   接收事件
   流程事件
   发送事件
}
完成
```

- 如果程序是由现有组件组成的，
  这些可以被“包装”以提供基于事件的接口
  - 这是 Façade 模式 (GoF) 的一个示例

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

表示事件流

- 为了允许通信和计算重叠（通常是为了避免序列化），您需要一种异步传递事件的方式

- 在消息传递环境中，事件可以通过从生成事件的任务异步发送到处理事件的任务的消息来表示

- 在共享内存环境中，可以使用共享队列来模拟消息传递

  – 我们将在课程后面介绍这种模式

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

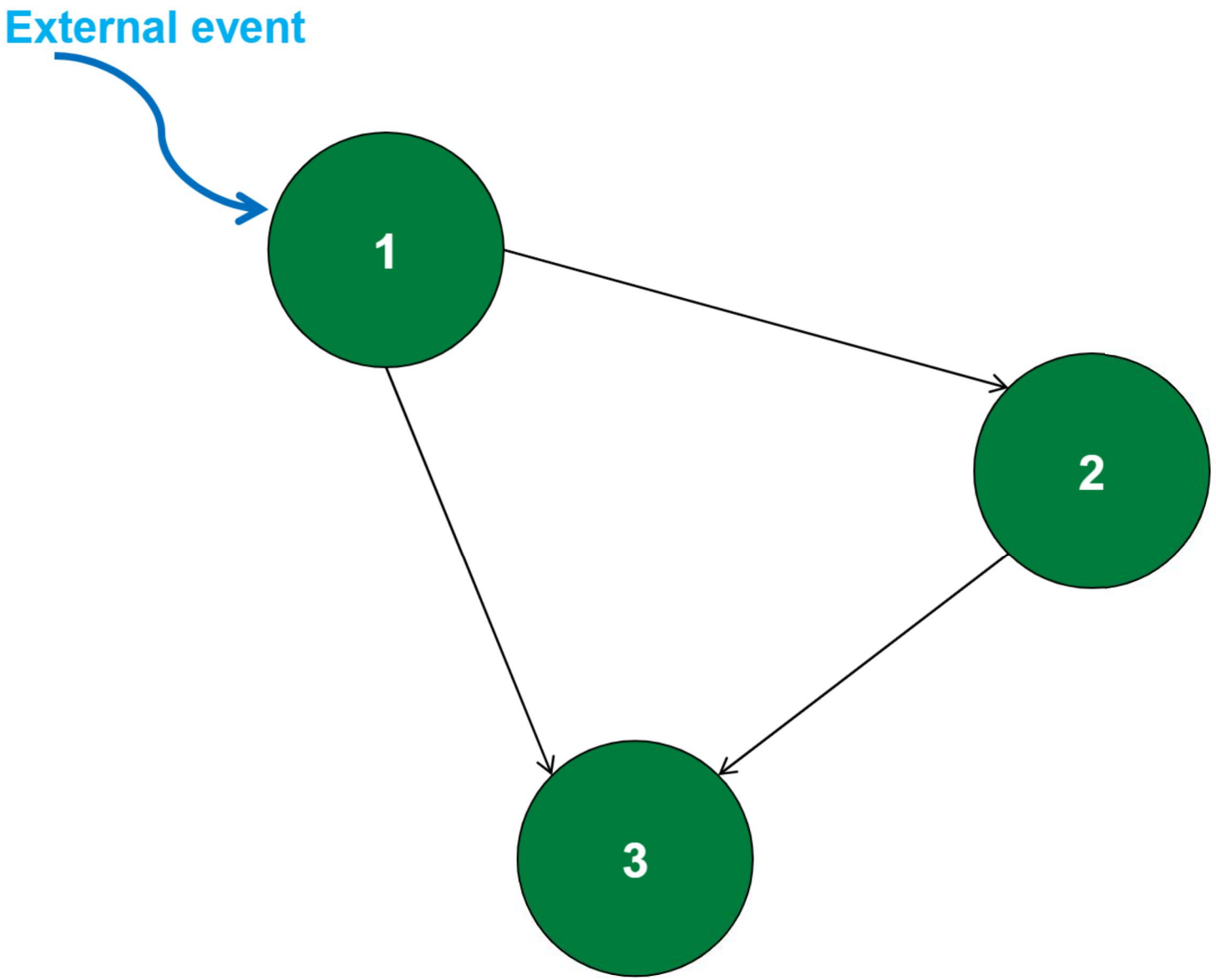
强制事件排序

- 这可能是应用此模式最困难的一步
- 强制执行排序约束可能会使任务必须按照与发送事件不同的顺序来处理事件

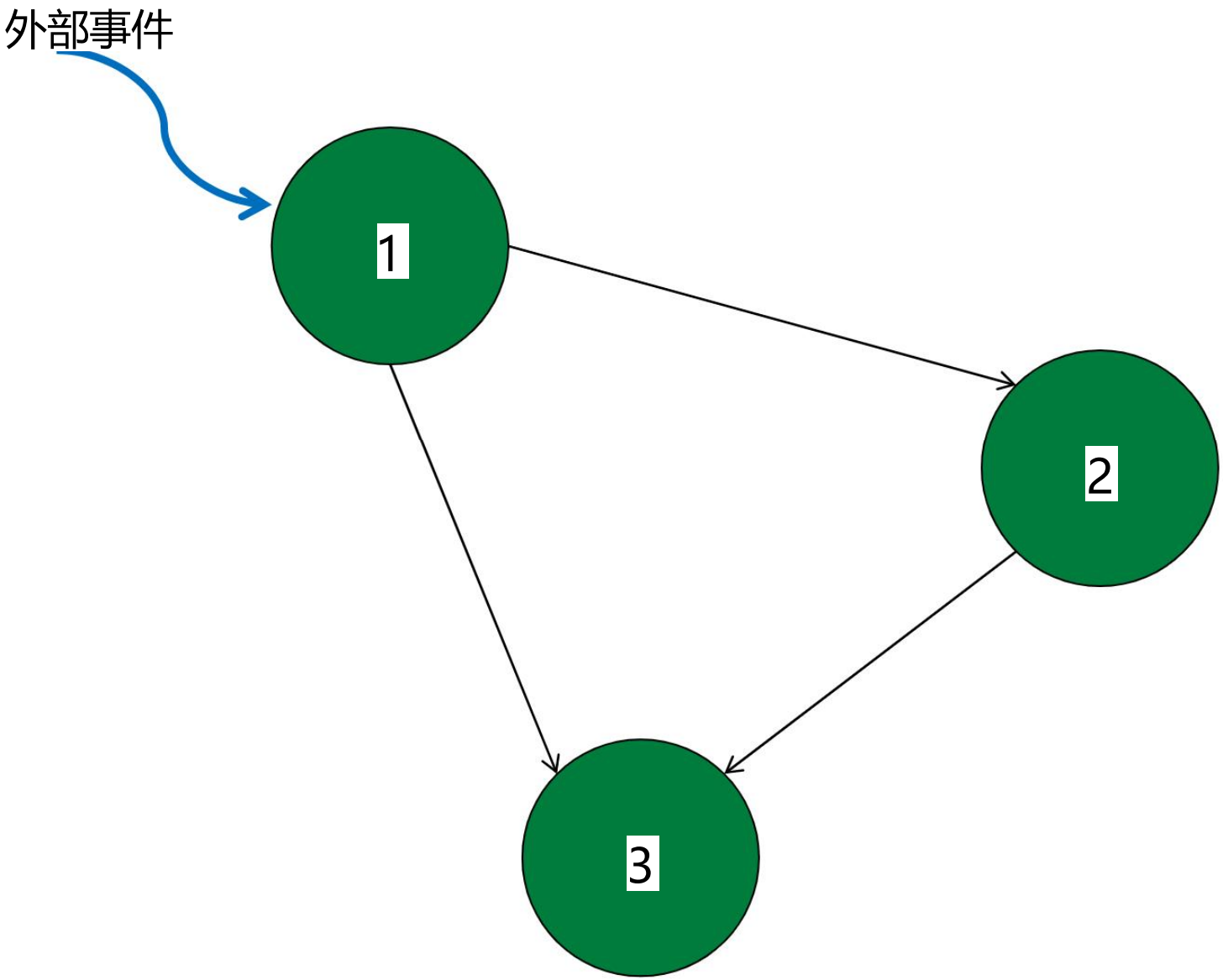
  – 或者，确实收到了。因此请注意，MPI 保证的 P2P 消息排序不一定足以保护您！——事件通常从不同的 UE 接收，这些 UE 可以按照任何顺序到达，因此也需要注意这里的任何限制。

- 因此，有时需要根据采取的方法，在事件队列中“向前看”确定正确的行为应该是什么

Event Ordering: Simple Example



事件排序：简单示例



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

事件排序

- 在花时间尝试强制执行事件排序之前，请检查它是否重要
  - 事件路径是线性的吗？
  - 应用程序是否会关心事件是否乱序？
- 如果确实重要，可以采取两类方法：
  - 乐观 – 继续，稍后处理问题，例如通过回滚 – 悲观 – 等待（例如等待同步事件）以确保顺序

约束得到满足，但这会产生开销

- 有助于解决事件顺序约束的机制
  - 全局时钟 – 同步事件

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)

避免死锁

- 这种模式的一个常见陷阱

- 你可以得到正常的消息传递死锁，但通过事件驱动的模拟，你也可能得到应用程序级死锁

  - 可能是实施错误，或“模型错误”

- 采用悲观的事件排序方法，过于悲观可能会导致死锁

  - 一些方法使用运行时死锁检测 – 作为一般解决方案通常效率很低
  - 超时可能会损害完整的死锁检测

    - （可就地处理）

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

调度和处理器分配

- 最直接的方法是每个 UE 一个
  - 允许所有任务同时执行
- 每个 UE 可以有多个任务
  - 效率可能不理想，但通常很难避免
- 使用此模式进行负载平衡可能很困难
  - 支持任务迁移的基础设施可以提供帮助

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

高效事件沟通

- 这种模式意味着大量的沟通
- 您需要一个高效的底层通信系统
  - 无论是基于消息还是基于共享内存
- 在消息传递环境中，可以合并消息（或有时拆分消息）以提高效率
- 在共享内存环境中，必须注意共享队列等的高效实现，因为这些很容易成为瓶颈

Event-Based Coordination: Summary

epcc

- 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

基于事件的协调：摘要

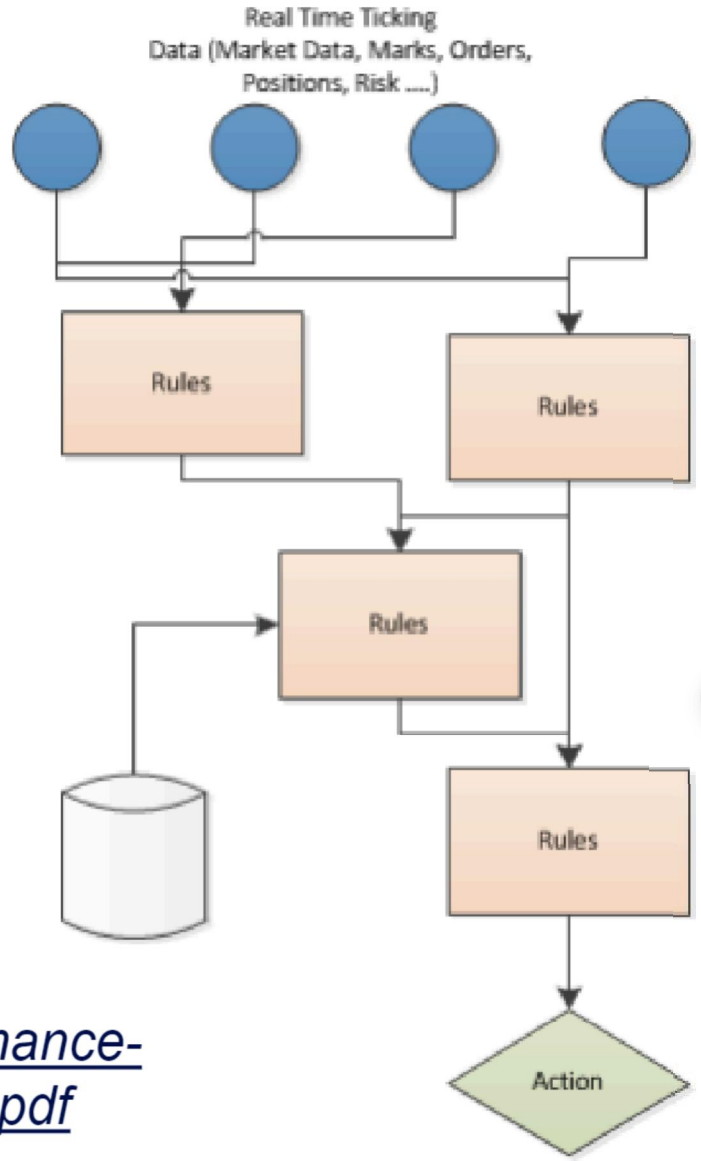
- 对于数据流不规则的问题
  - 互动的不可预测性
- 将现实世界的实体映射到任务
- 通过事件模拟现实世界的互动
- 最难做到的往往是事件的排序，因为沟通不是即时的，而现实世界

  你建模的交互通常是同步的
- 可以灵活地改变模型的细节，而无需改变并行化策略/代码

- 可以应用于现有的代码组件

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

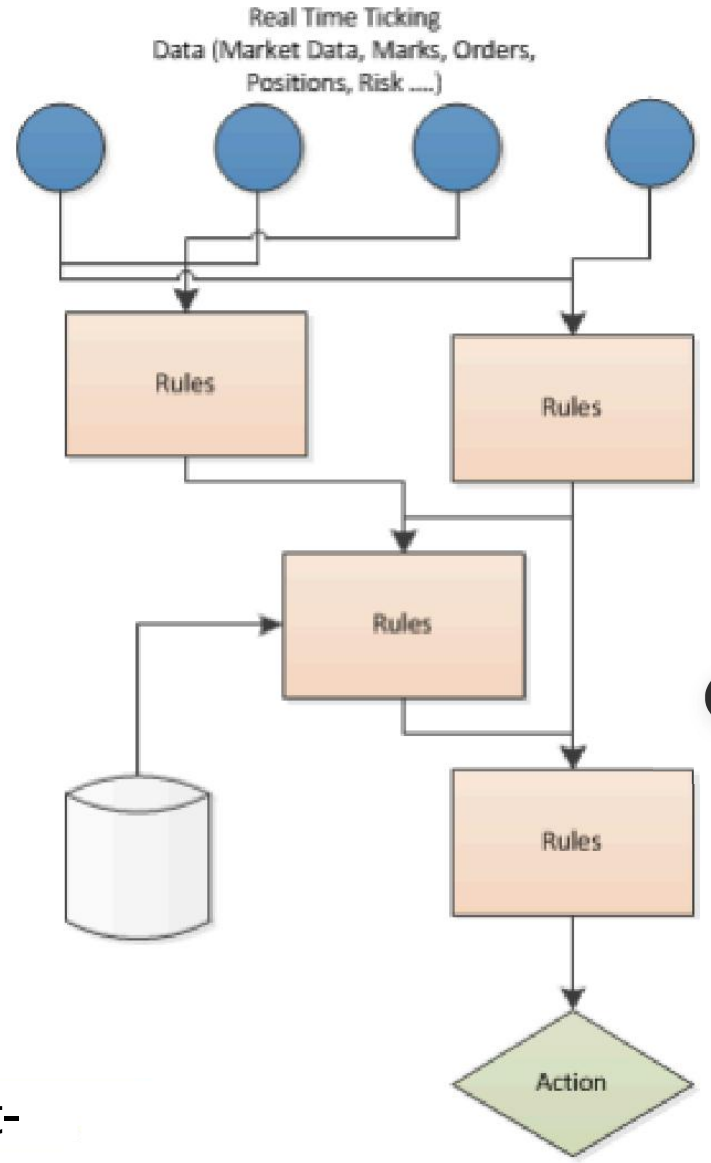


http://www.bcs.org/upload/pdf/application-of-high-performance-and-low-latency-computing-in-investment-banks-080115.pdf

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

瑞银金融信息交换

- 实时市场数据
  - – 用于报价、订单和执行 – 峰值可达每秒 1600 万项（带宽） – 需要在几毫秒内处理事件（延迟）



- 各阶段同时进行
  - – 通常每个都在自定义硬件上的单独线程中运行 – 讨论将“计算放在数据平面中”
- 低级优化
  - – 在网络、操作系统和运行时级别 – 基于 FPGA 的硬件

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

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