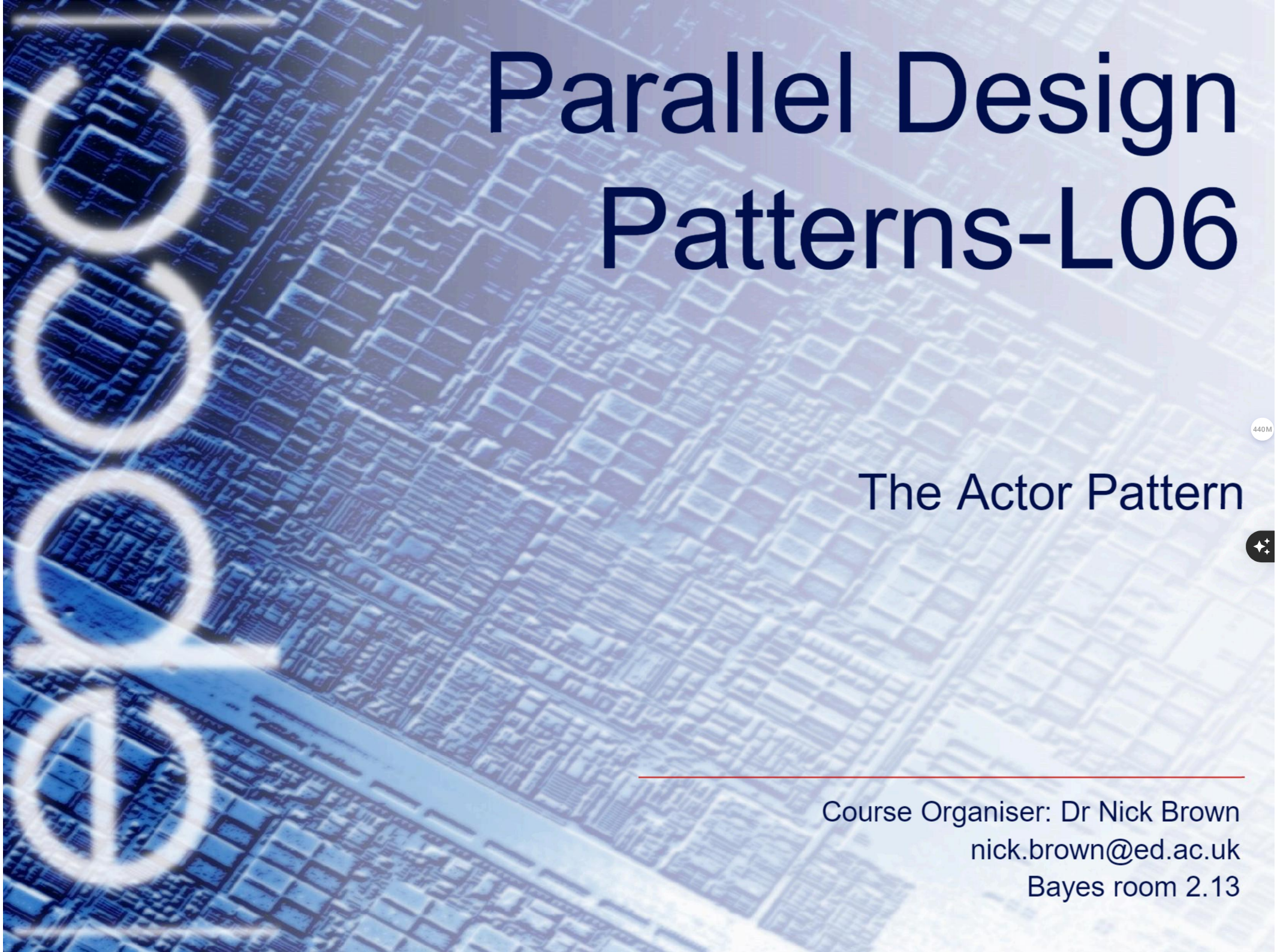
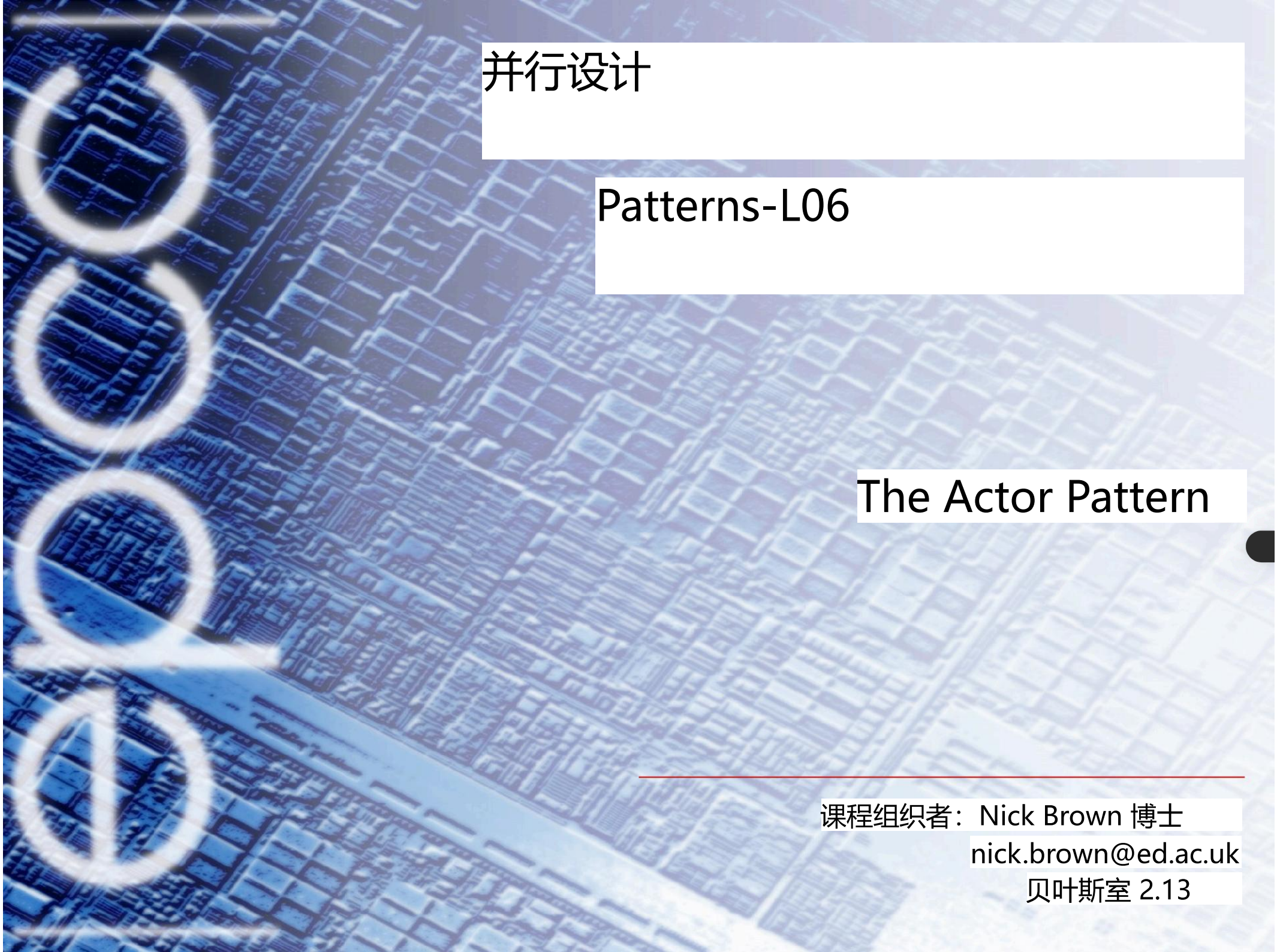


Parallel Design Patterns-L06

The Actor Pattern

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



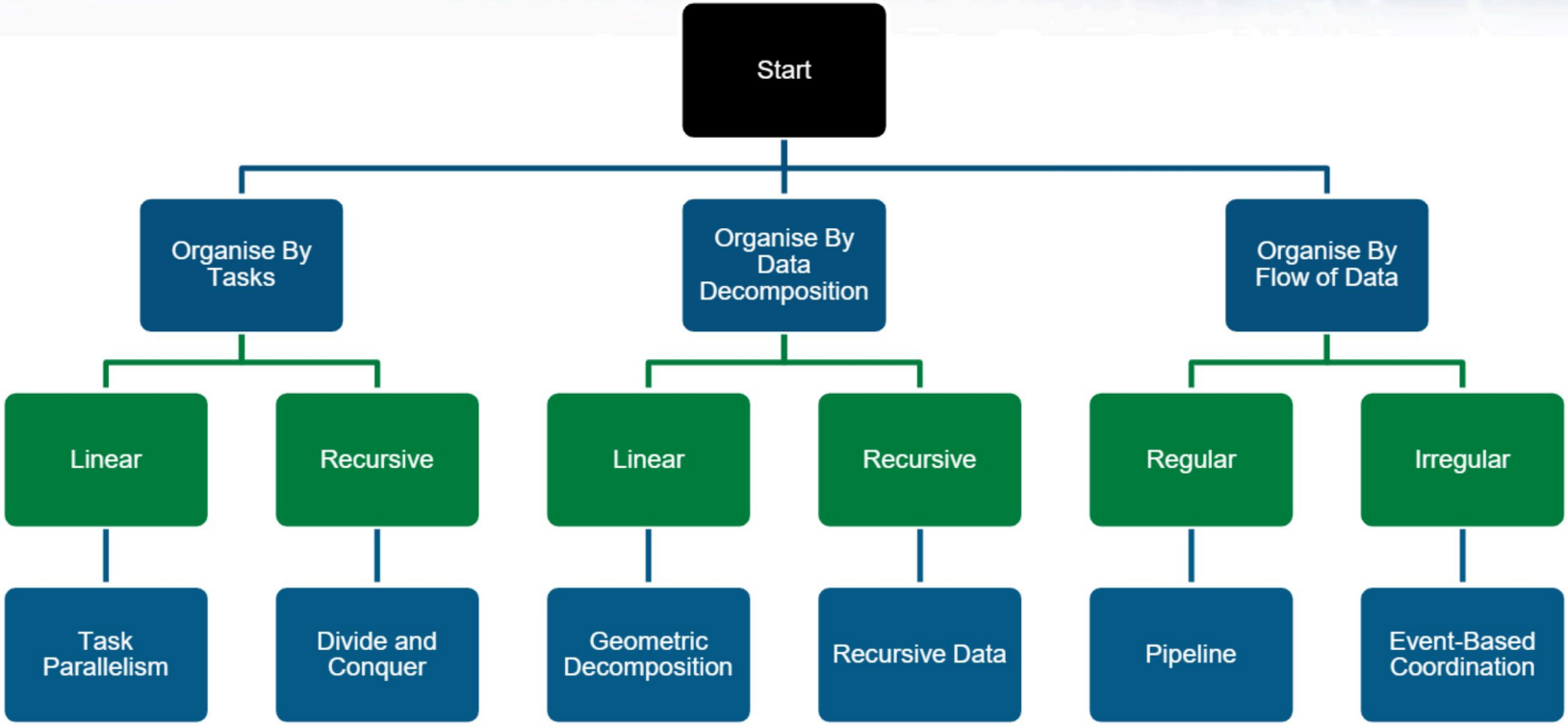
并行设计

Patterns-L06

The Actor Pattern

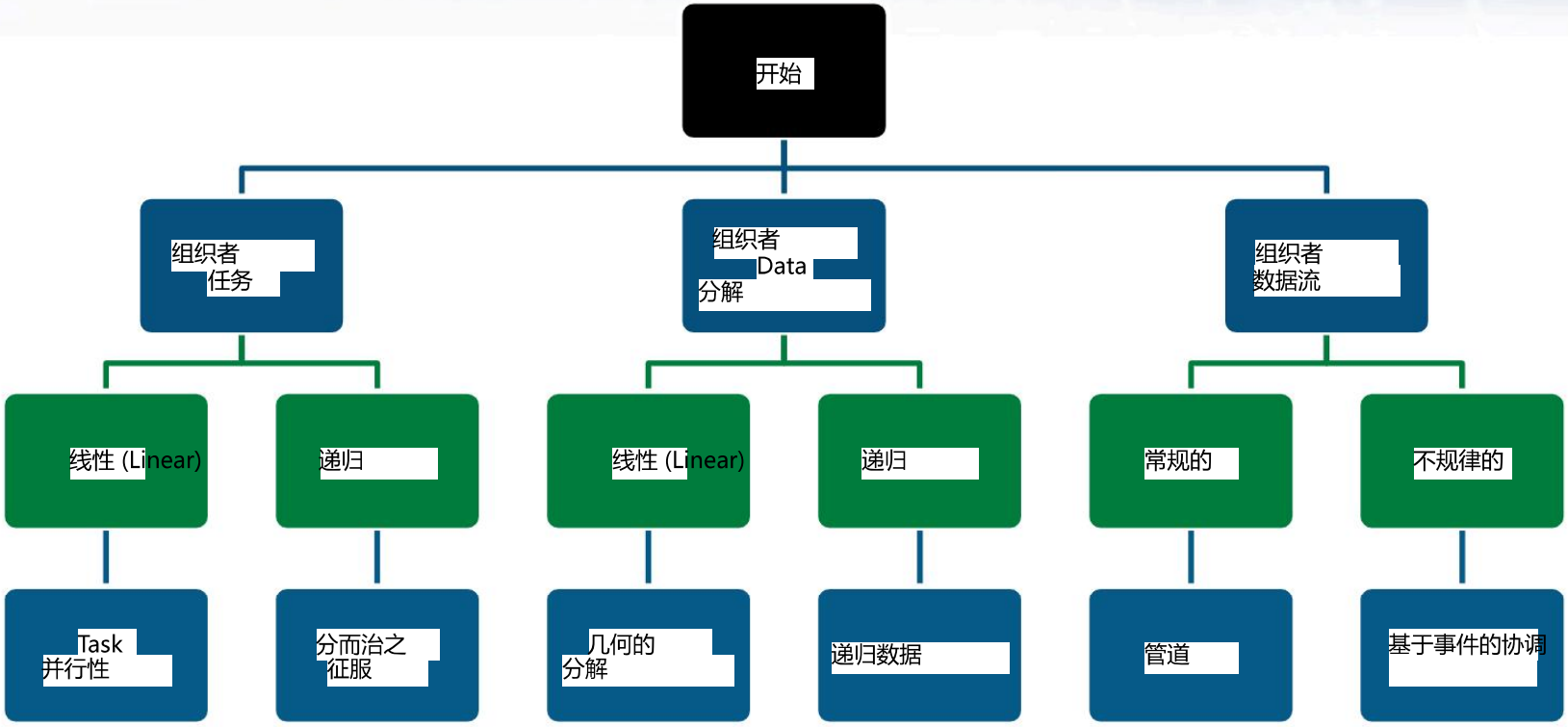
课程组织者：Nick Brown 博士
nick.brown@ed.ac.uk
贝叶斯室 2.13

The Actor Pattern



- The Actor Pattern is a *Parallel Design Pattern* in the *algorithm strategy/parallel algorithm structure* space
- It does not fit directly into the classification of *Mattson* et al
  - It is **closely related** to Event Based Coordination
  - Major organising feature of the actor pattern is where parallelism (the number of tasks and their interactions) are dynamic and unpredictable

Actor 模式



- Actor 模式是

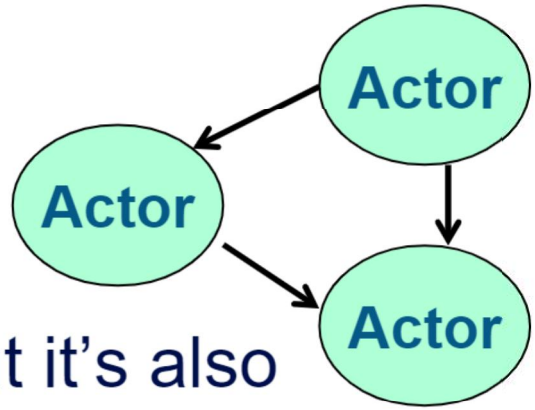
  算法策略/并行算法结构空间

- 它并不直接符合 Mattson 等人的分类

  – 它与基于事件的协调密切相关 – 参与者模式的主要组织特征是并行性（任务及其交互的数量）是动态的和不可预测的

The Actor Pattern

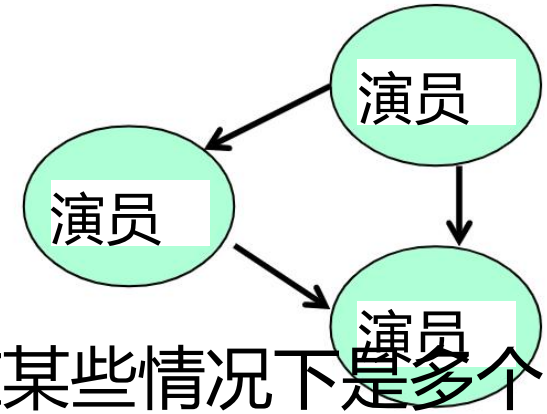
- Like event-based co-ordination, the solution is to map real-world entities on to tasks with a 1:1 mapping



- The mapping of tasks to UEs is *often* 1:1, but it's also quite common to map several (in some cases, *many*) tasks to each UE
- Conceptually the model is of independent actors interacting *only* through the exchange of messages
  - Actor pattern uses the terminology "message"
  - A "message" is like an event, and it has an intended recipient (another actor)
  - Very similar to an MPI message but note that an MPI message is between *processes* and an Actor message is between *actors*

Actor 模式

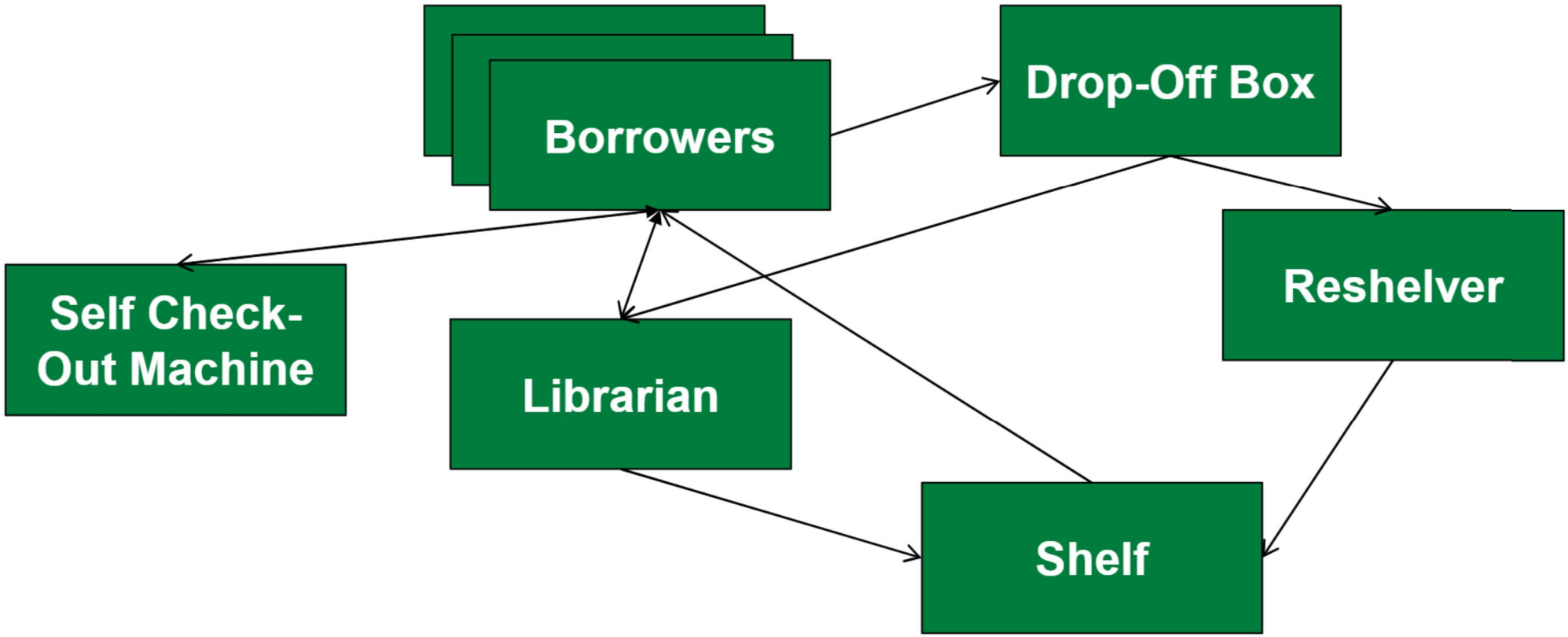
- 与基于事件的协调一样，解决方案是将现实世界的实体以 1:1 的映射关系映射到任务上



- 任务与 UE 的映射通常是 1：1，但将多个（在某些情况下是多个）任务映射到每个 UE 也很常见

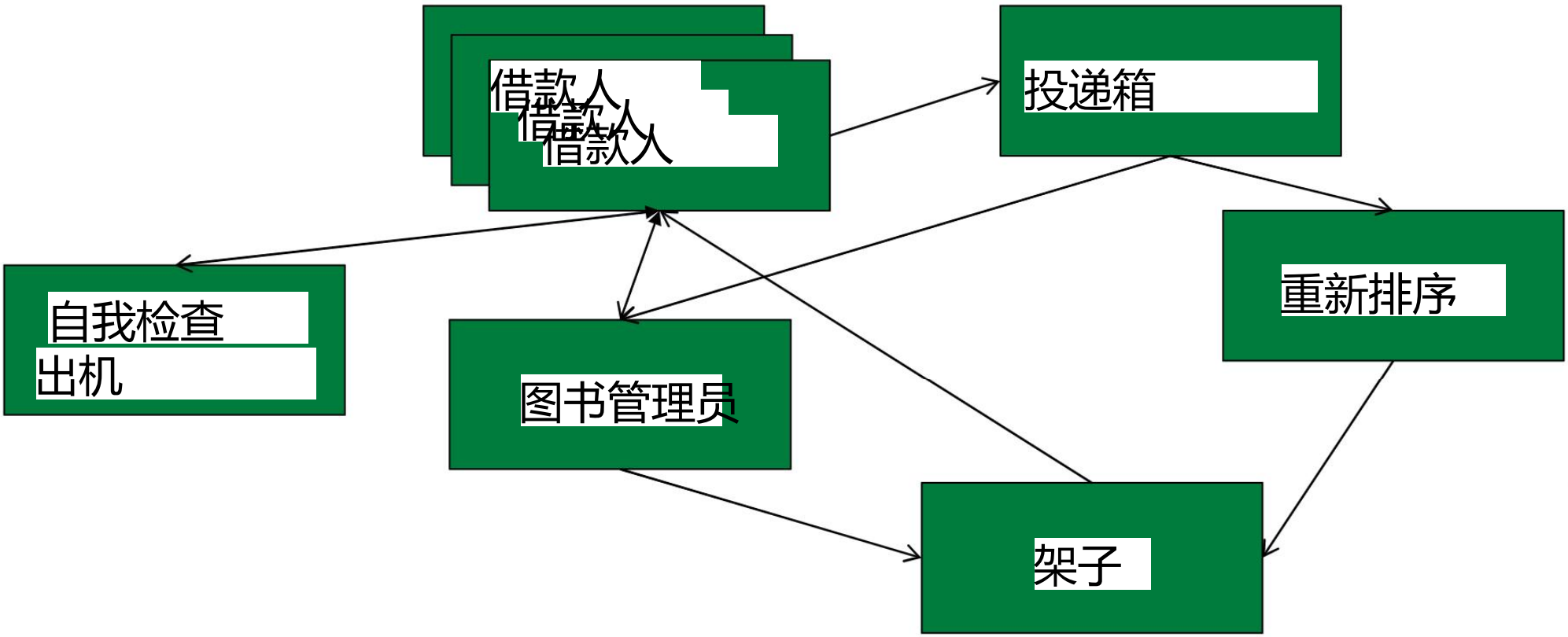
- 从概念上讲，该模型由独立的参与者组成，他们仅通过交换信息进行交互
  - 参与者模式使用术语“消息”
  - “消息”就像一个事件，它有一个预期的接收者（另一个参与者）——与 MPI 消息非常相似，但请注意，MPI 消息是在进程之间，而 Actor 消息是在参与者之间

Contrast with Event based coordination



- Number of actors (tasks) is dynamic, as borrowers come and go
- Actors can perform their own work that is not driven by any messages

与基于事件的协调形成对比



- 随着借款人的来来去去，参与者（任务）的数量是动态的
- Actor 可以执行自己的工作，不受任何消息的驱动

Everything is an Actor

- The Actor Pattern “philosophy” is:

Everything is an Actor

- In the same spirit to
  - “everything is an object” in OO programming
  - “everything is a file” in UNIX

- This is a *way of thinking* about your problem

一切都是演员

- 演员模式的“哲学”是：

一切都是演员

- 本着同样的精神
  - OO 编程中“一切皆对象” – UNIX 中“一切皆文件”
- 这是思考问题的一种方式

Everything an actor?

- In fact, just like with all patterns, the Actor pattern can be combined with other patterns
  - …but don't use non-actor components just because they're more familiar to you
  - Try to think within the actor pattern and **make everything an actor**

- Why make everything an Actor?
  - Maintains a symmetry
    - All elements in the program can interact in the same way: Through messages
    - Don't need to add the complication of how actors communicate with non-actors
    - As soon as we start to add none actors then loose some of the advantages of this model

一切都是演员？

- 事实上，就像所有模式一样，Actor 模式可以与其他模式结合

  – ...但不要仅仅因为它们更
  熟悉的——尝试在演员模式中思考，让一切都成为演员

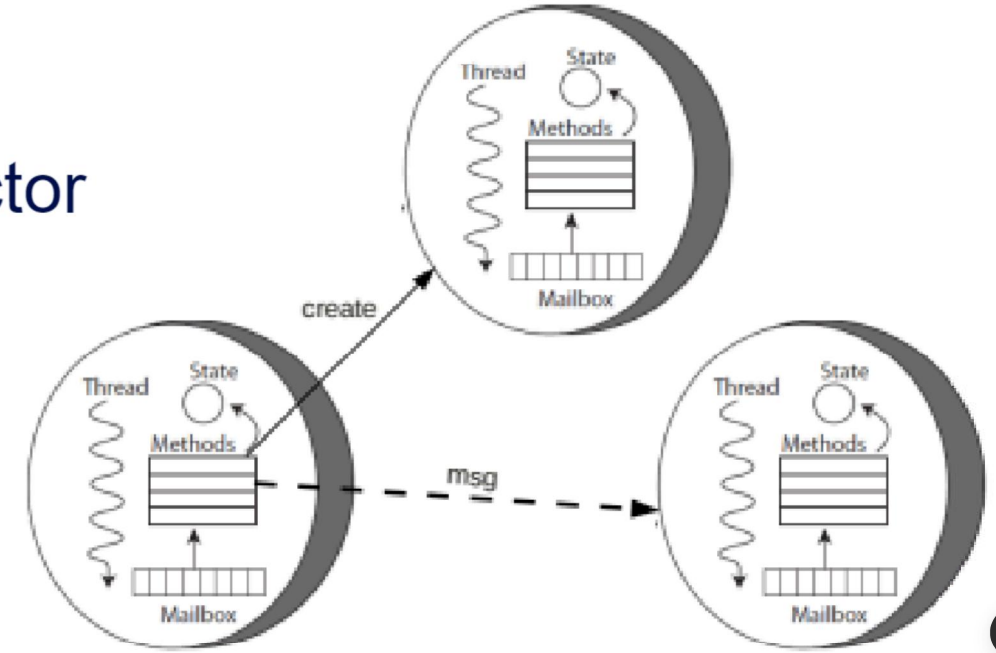
- 为什么要把所有东西都变成 Actor?
  – 保持对称性 – 程序中的所有元素都可以以相同的方式交互：通过

    消息

  – 不需要增加参与者沟通的复杂性
  与非参与者一起——一旦我们开始添加非参与者，就会失去此模型的
  一些优势

An actor can…

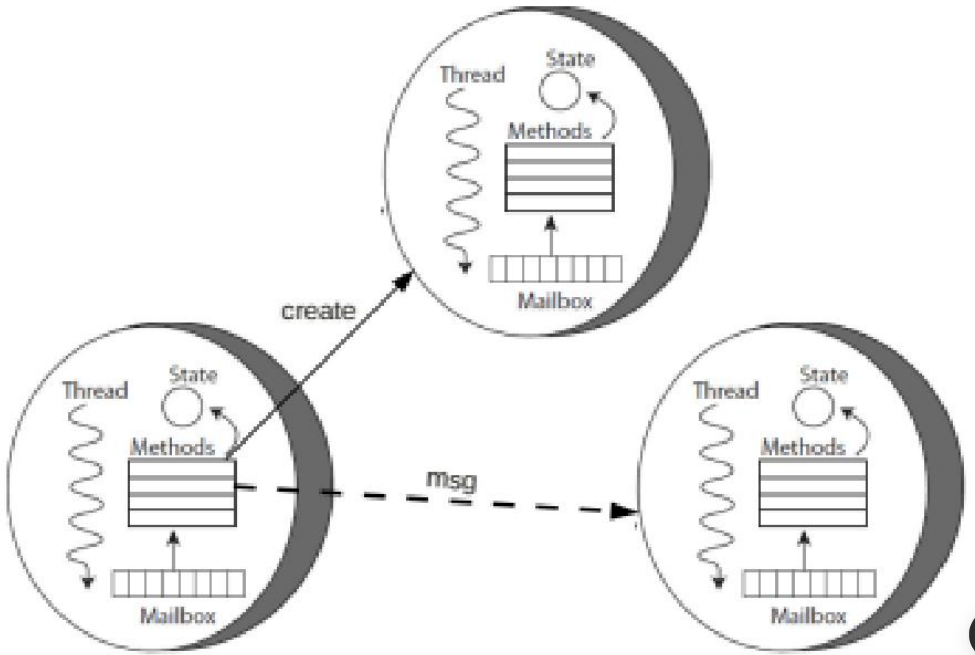
- Receive a message from any actor
- Do computational work
- Send a message to another actor
- Create a new actor
- Die



- It is entirely up to a specific actor to decide how to respond to a message from another and different actors might very well respond in different ways
- They maintain their own state

演员可以……

- 接收来自任何演员的消息
- 进行计算工作
- 向另一位演员发送消息
- 创建新演员
- Die



- 如何响应来自另一个参与者的消息完全由特定的参与者决定，并且不同的参与者可能会以不同的方式做出回应

- 他们维持着自己的国家

An actor should

- Be perfectly encapsulated, ideally there should not be any shared state between them

- Represent and be anything
  - Within an actor can still use other parallel patterns to help with computational work

- Fit in well with the idea of a framework
  - A framework could provide the underlying mechanisms for communication, scheduling etc
  - The programmer provides their own actor (or actor behaviour) to flesh this out and specialise it for their own application
  - One actor doesn't need to care about what other actors are doing (apart from understanding their messages.)

演员应该

- 完美封装，理想情况下它们之间不应该有任何共享状态

- 代表并成为任何东西
  - 参与者仍然可以使用其他并行模式来帮助完成计算工作

- 非常符合框架的理念
  - 框架可以提供用于通信、调度等的底层机制 – 程序员提供他们自己的参与者（或参与者行为）来充实它并使其专门用于他们自己的应用程序 – 一个参与者不需要关心其他参与者在做什么（除了理解他们的消息。）

Things that can be actors

- Objects representing *particles*, *people, animals, books,* or *any real-life entity*
- Grid cells
  - an alternative to domain decomposition
  - useful, for example, when the actual geometry is less important and the main interaction with the grid cell is with other actors
- Global features / fields
  - Actors don't have to represent something localised in space
- Clocks
  - Actors generally act asynchronously and out-of-lockstep
  - It is sometimes useful to have some global notion of time which can be implemented by a clock Actor which sends messages to those Actors that need to be aware of global time
  - Time is often coarse grained without notion of computational steps

可以充当演员的东西

- 代表粒子、人、动物、书籍或任何现实生活中的实体的对象

- 网格单元
  - 域分解的替代方法 – 例如，当实际几何形状不太重要并且与网格单元的主要交互是与其他参与者交互时很有用

- 全局特征/字段
  - 参与者不必代表空间中某个特定的地方
- 时钟
  - Actor 通常以异步和不同步的方式行动 – 有时需要一些全局的时间概念，可以通过时钟 Actor 来实现，它会向需要了解全局时间的 Actor 发送消息
  - 时间通常是粗粒度的，没有计算步骤的概念

Benefits of the actor pattern

epcc

- Very flexible
  - As anything can be an actor then (theoretically) we can use this to model just about anything
  - If your system contains a number of different types of entities that need to interact then this can be helpful
- Actors encompass the ideas of modularity and encapsulation
  - As they are self contained and atomic, it should be trivial to add new types of actors
  - Do need a way to ensure that other actors can deal with messages from them
- Embrace the chaotic and unpredictable nature of parallelism
  - Other patterns attempt to control this using synchronisation and constrains, the actor pattern supports us managing it

演员模式的好处

- 非常灵活
  - 因为任何事物都可以成为参与者，所以（理论上）我们可以用它来模拟任何事物 – 如果你的系统包含许多需要交互的不同类型的实体，那么这可能会有所帮助
- Actor 包含了模块化和封装的思想
  - 由于它们是自包含且原子的，因此添加新类型的参与者应该很简单 – 确实需要一种方法来确保其他参与者可以处理来自他们的消息
- 拥抱并行性的混乱和不可预测的本质
  - 其他模式尝试使用同步和约束来控制这一点，而参与者模式支持我们管理它

Criticisms of the actor pattern

- There is often less order to the system
  - How do we do effective load balance if the actors have different computational requirements?
  - As actors can create other actors dynamically the state of the system can change dramatically and unpredictably.

- Can involve many messages flowing around unpredictably
  - Hard to design any locality into communication
  - Need to be careful when it comes to message ordering (as in the event based coordination model)
  - Unbounded nondeterminism, where the delay in servicing a message can appear to have no limit whilst still guaranteeing that the request will eventually be serviced.

对 Actor 模式的批评

- 系统通常不太有序
  - 如果参与者有不同的计算要求，我们如何实现有效的负载平衡？ – 由于参与者可以动态地创建其他参与者，因此系统的状态可能会发生巨大且不可预测的变化。

- 可能涉及许多不可预测的消息流动
  - 很难在通信中设计任何局部性 – 涉及消息排序时需要小心（如基于事件的协调模型） – 无界的不确定性，其中服务消息的延迟似乎没有限制，同时仍保证请求最终将得到服务。

ACTOR PATTERN & MPI

参与者模式和 MPI

Programming Languages & Libraries

epcc

- ABCL
- AmbientTalk
- Axum
- E
- Erlang
- Fantom
- Humus
- Io
- Ptolemy Project
- Rebeca Modeling Language
- Reia
- Rust
- SALSA
- Scala
- Scratch

- Akka
- Ateji PX
- F# MailboxProcessor
- Korus
- Kilim
- ActorFoundry (based on Kilim)
- ActorKit
- Retlang
- Jetlang
- Haskell-Actor
- GPars (was GParallelizer)
- PARLEY
- Pykka (inspired by Akka)
- Termite Scheme
- Theron
- Libactor
- Actor-CPP
- S4
- libcppa

Source: Wikipedia (http://en.wikipedia.org/wiki/Actor_model)

编程语言和库

- ABCL
- 环境谈话
- Axum
- E
- Erlang
- 幻影
- 腐殖质
- Io
- 托勒密计划
- Rebeca 建模语言
- Reia
- Rust
- 萨尔萨舞
- Scala
- 划痕
- Akka
- 阿替吉 PX
- F# 邮箱处理器
- 科鲁斯
- 基里姆
- ActorFoundry（基于 Kilim）
- ActorKit
- 雷特朗
- 捷特朗
- Haskell-Actor
- GPars（原为 GParallelizer）
- 帕利
- Pykka (受 Akka 启发)
- 白蚁防治计划
- 塞隆
- 利巴克托
- Actor-CPP
- S4
- 库库

资料来源：维基百科 (http://en.wikipedia.org/wiki/Actor_model)

- MPI is **not** a perfect fit for the Actor pattern but when used in the right way, MPI can be used to implement the Actor pattern
  - What's more, if you want to run an Actor Pattern-based code on a massively parallel machine, you probably don't have a lot of choice
    - or at least, your other choices just have different shortcomings

- Harder things to do with MPI:
  - Creation and destruction of Actors
  - Fire-and-forget asynchronous messages

- Unnecessary aspect of MPI:
  - Program looks like it's SPMD. All actors have to start off running the same program

- MPI 并不完美适合 Actor 模式，但如果使用得当，MPI 可用于实现 Actor 模式
  - 更重要的是，如果你想在大规模并行机器上运行基于 Actor 模式的代码，你可能没有太多选择
    - 或者至少，你的其他选择只是有不同的缺点

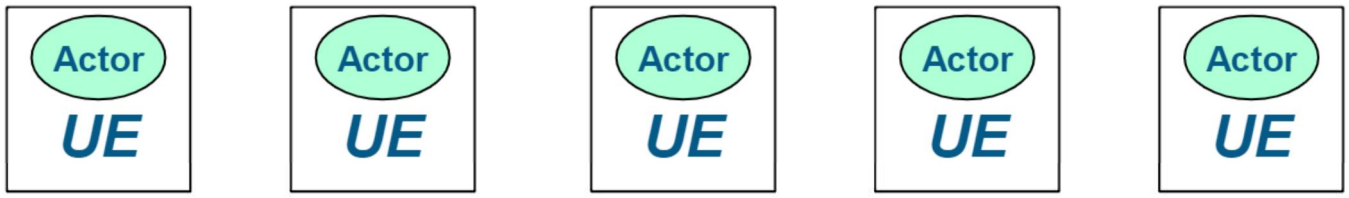
- 使用 MPI 做起来比较困难的事情：
  - Actor 的创建和销毁 – 即发即弃的异步消息

- MPI 不必要的方面：
  - 程序看起来像是 SPMD。所有参与者必须首先运行同一个程序

One Actor per UE

- Advantages
  - Conceptually more simple
  - Exposes most parallelism
  - Actor messages map directly to MPI messages
  - It's possible (although not always desirable) to use MPI ranks to index the actors

- Disadvantages
  - Might require a very large number of UEs
  - Load balancing might become an issue



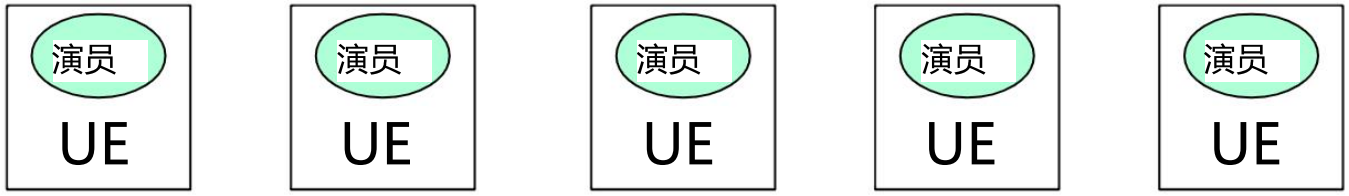
每个 UE 一个 Actor

- 优点

  – 概念上更简单 – 公开大多数并行性 – Actor 消息直接映射到 MPI 消息 – 可以（尽管并不总是可取）使用 MPI 等级来索引 Actor

- 缺点

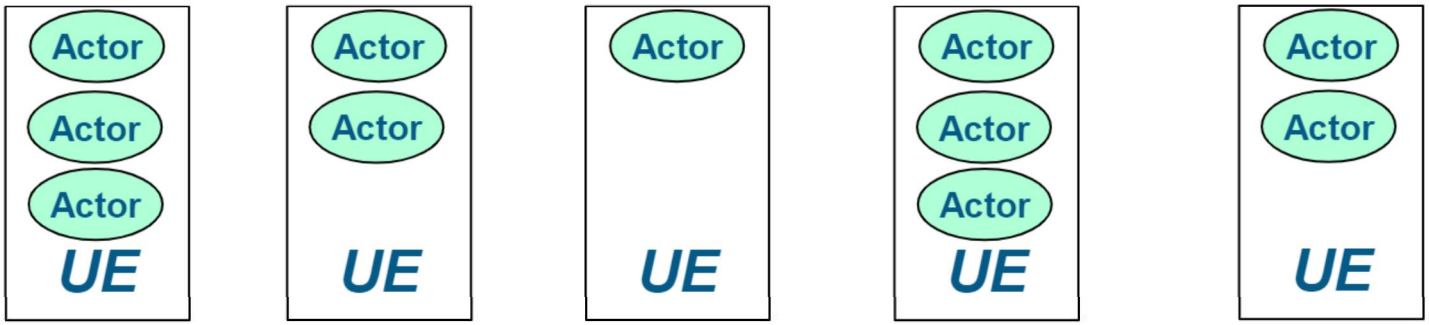
  – 可能需要大量 UE – 负载平衡可能成为一个问题



Multiple Actors per UE

- Advantages
  - Less low level parallel overhead (number of UEs can match target architecture)
  - Actor creation and destruction is simpler as don't need to create any UEs
  - Can mix actors with different computational requirements

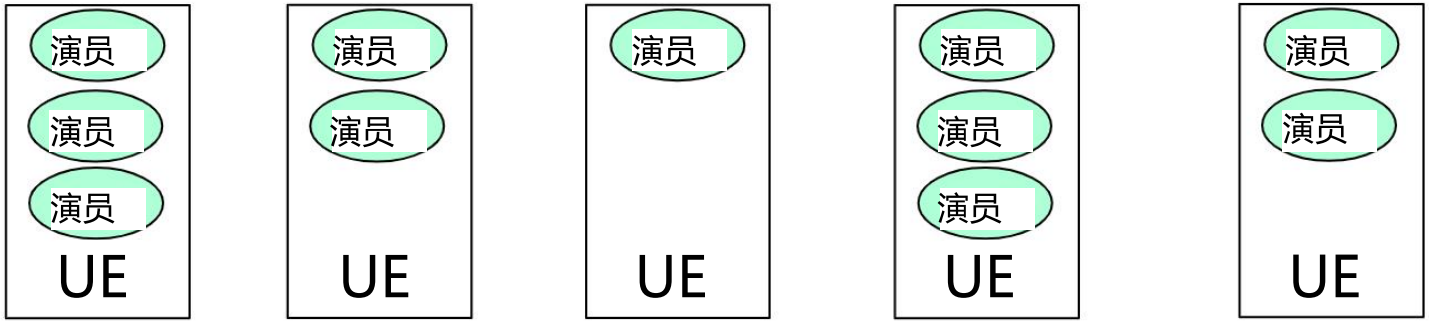
- Disadvantages
  - You loose symmetry (actors on the same UE (local) as well as remote actors can communicate)
  - Need to provide your own messaging solution, such as an event queue for each UE.



每个 UE 有多个 Actor

- 优点
  - 更少的低级并行开销（UE 数量可以与目标架构匹配） – 参与者的创建和销毁更简单，因为不需要创建任何 UE – 可以混合具有不同计算要求的参与者

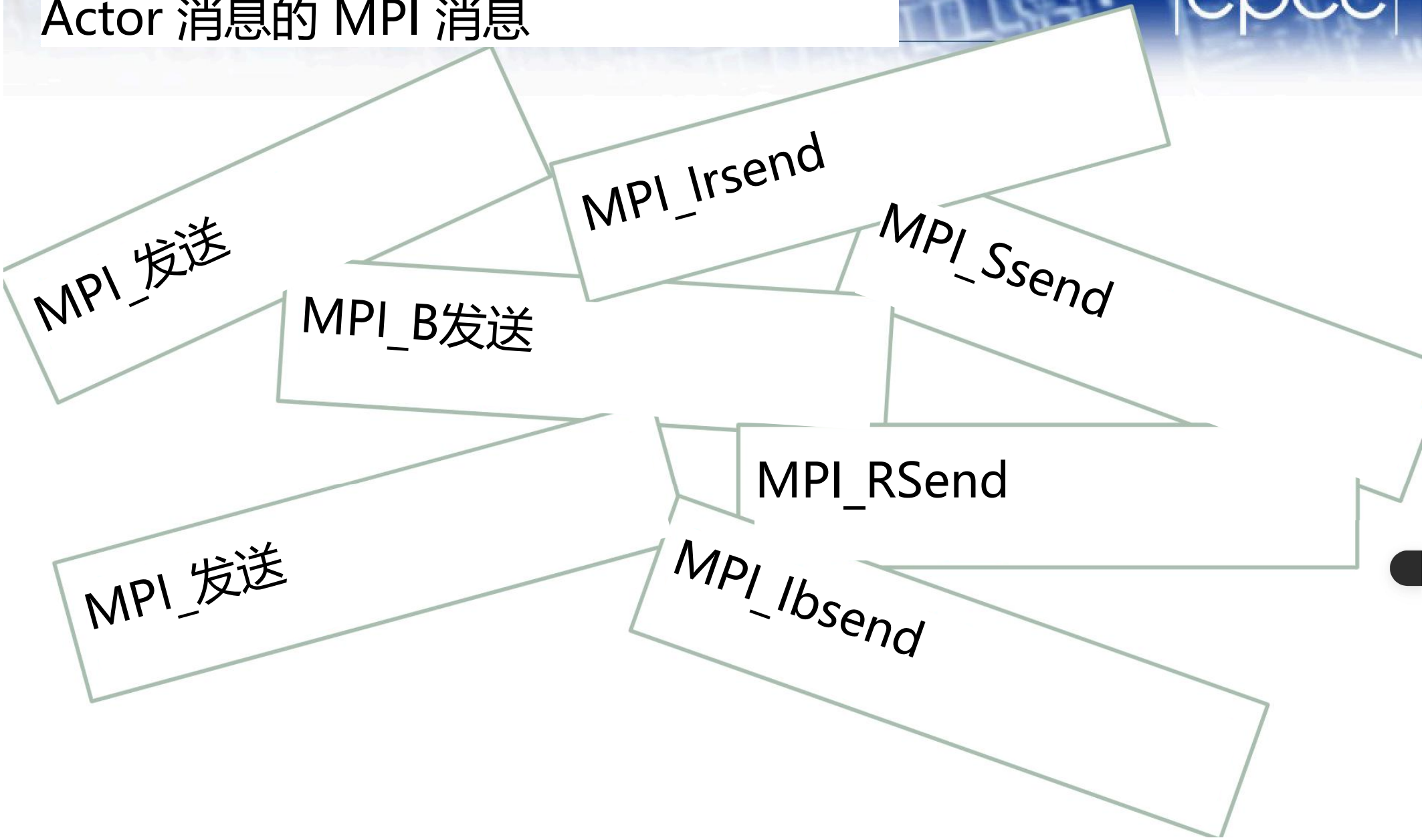
- 缺点
  - 您失去对称性（同一 UE（本地）上的参与者以及远程参与者可以通信） – 需要提供您自己的消息传递解决方案，例如每个 UE 的事件队列。



MPI messages for Actor Messages

- What kind of MPI message is it best to use to represent an Actor message?

Actor 消息的 MPI 消息



- 最适合使用哪种 MPI 消息来表示 Actor 消息?

Buffered Sends for Actor Messages

epcc

- None are perfect but the best one to is a **buffered send**, as this is the closest fit to *fire-and-forget*

- `int` **`MPI_Bsend`**`(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - **buf** initial address of send buffer (choice)
  - **count** number of elements in send buffer (nonnegative integer)
  - **datatype** datatype of each send buffer element (handle)
  - **dest** rank of destination (integer)
  - **tag** message tag (integer)
  - **comm** communicator (handle)

Actor 消息的缓冲发送

- 没有一种是完美的，但最好的是缓冲发送，因为这是最接近发射后不管的

- int MPI_Bsend(void *buf, int count,
  MPI_Datatype 数据类型，int 目标，int 标签，
  MPI_Comm 通信)

  – buf 发送缓冲区的初始地址（选择） – count 发送缓冲区中的元素数（非负整数） – datatype 每个发送缓冲区元素的数据类型（句柄） – dest 目标的等级（整数） – tag 消息标签（整数） – comm 通信器（句柄）

How buffered sends work

- MPI_Bsend causes the contents of *data* to be copied into an internal MPI buffer

- As soon as the contents of *data* have been copied, the call completes and the process moves on to next line in the program
  - You can then re-use / modify *data* without the message being affected
  - It is not guaranteed that the message has been received by the receiver, and there's no way to check that the message has been received

缓冲发送的工作原理

- MPI_Bsend 导致数据内容被复制到内部 MPI 缓冲区中

- 一旦数据内容被复制，调用就完成，并且进程转到程序的下一行

  – 然后，您可以重新使用/修改数据，而不会影响消息 – 不能保证该消息已被接收

    接收者，并且没有办法检查消息是否已经已收到

MPI_Buffer_attach

- The programmer must specify the size of the buffer with MPI_Buffer_attach

- `int MPI_Buffer_attach(void *buffer, int size)`
  - **buffer** initial buffer address (choice)
  - **size** buffer size, in bytes (integer)

- If the buffer is full, your program will error
  - For an actor pattern which could have many messages in transit, you should probably start with a large buffer size (allowing, say, hundreds of messages to be buffered)

MPI_Buffer_attach

- 程序员必须使用 MPI_Buffer_attach 指定缓冲区的大小

- `int MPI_Buffer_attach(void *缓冲区, int 大小)`
  - 缓冲区初始缓冲区地址（选择） – 大小
    缓冲区大小，以字节为单位（整数）

- 如果缓冲区已满，你的程序将会出错
  对于可能传输许多消息的 Actor 模式，你应该从一个较大的缓冲区大小开始（例如，允许缓冲数百条消息）

What should I send?

- Contents of the buffer could, in general, be some kind of message structure
- In practice, if your application allows it, it can be far simpler to use a known (basic) data type, with a known count
  - For example, if you know that the only data that needs to be included with any of your messages is of integer type, and you know that you'll never need to send more than two integers in a message, you can also use an integer to define your message type and just make all of your sends and receives of the form

    - `MPI_Bsend(data, 3, MPI_INTEGER, …)`
    - `MPI_Irecv(data, 3, MPI_INTEGER, …)`

Where integer 1 is the command/type and 2 & 3 are data associated with it

我应该发送什么？

- 缓冲区的内容通常可以是某种消息结构

- 实际上，如果你的应用程序允许，使用已知（基本）数据类型和已知计数会简单得多

  – 例如，如果你知道唯一需要包含的数据
  您的任何消息都是整数类型，并且您知道您将
  永远不需要在一条消息中发送超过两个整数，你也可以使用整数来定义你
  的消息类型，并使所有的发送和接收都具有以下形式

  – MPI_Bsend(数据，3，MPI_INTEGER，...) –
  MPI_Irecv(数据，3，MPI_INTEGER，...)

其中整数 1 是命令/类型，2 和 3 是与其相关的数据

Receiving Messages

- MPI requires matching sends and receives
- The actor pattern requires that an actor can get on with doing what it is doing and not have to wait for messages, we have two choices (I find the second tends to be simpler):
  - **MPI_Irecv** but the downside is keeping track of request handles and cancelling this in termination
  - **MPI_Iprobe** to check for messages and then **MPI_Recv** if a message is outstanding can be simpler – as no request handles.
- Since buffered sends are used, MPI messages can queue up, so for a simple implementation you only need to post one receive at a time
  - If you need to “look ahead” in your queue, you might want more

接收消息

- MPI 需要匹配的发送和接收
- 演员模式要求演员可以继续做它正在做的事情而不必等待消息，我们有两种选择（我发现第二种更简单）：
  - MPI_Irecv 但缺点是要跟踪请求句柄并在终止时取消它 – MPI_Iprobe 检查消息，然后 MPI_Recv 如果有消息未完成，这会更简单 – 因为没有请求句柄。
- 由于使用缓冲发送，MPI 消息可以排队，因此对于简单的实现，您只需一次发布一个接收
  - 如果你需要“提前查看”你的队列，你可能需要更多

An Actor in MPI

Simple codes just wait here for a message

```
do {
  MPI_Irecv(message, …, request)
  while not MPI_Test(request,…){
      do_compute_work_step()
  }
  process(message)
}
```

```
do {
  MPI_Iprobe(…, outstanding, status)
  if (!outstanding) {
        do_compute_work_step()
  } else {
     MPI_Recv(message,
     process(message, …, status.MPI_SOURCE, …)
  }
}
```

- Both *do_compute_work_step* and *process* functions could include **MPI_Bsend**s to send off new messages

- If *process* is time consuming, it could just add *message* to a local message queue to be handled during *do_compute_work_step*

MPI 的一名演员

简单的代码只需在这里等待消息

```
do {
   MPI_Irecv(消息，…，请求) 而非
   MPI_Test(请求，…){
do_compute_work_step（） } 过程（消息） }
```

```
do {
   MPI_Iprobe（…，未完成，状态）
   如果 (!未完成) {
   do_compute_work_step() } else {
   MPI_Recv(消息，

进程（消息，…，状态。MPI_SOURCE，…） } }
```

- Both
  执行计算工作步骤
  处理函数可以包括
  MPI_Bsends 来发送新消息

- 如果该过程耗时，它可以将消息添加到本地消息队列，以便在

  执行计算工作步骤

Receiving data from anyone

- Remember that an actor might (unpredictably) receive data from any other actor
  - It is therefore common to use *MPI_ANY_SOURCE* in place of a receiver's explicit process id
  - Can have a look at the MPI status to figure out the pid of the sender

```
MPI_Request request;
MPI_Status status;

MPI_Irecv(&message, 3, MPI_INT,
        MPI_ANY_SOURCE, …,
        &request)
MPI_Wait(&request, &status);

int source=status.MPI_SOURCE;
```

```
MPI_Status status;
int outstanding;

MPI_Iprobe(MPI_ANY_SOURCE, …,
        outstanding, &status)
if(outstanding) {
  int source=status.MPI_SOURCE;
  ……
}
```

In Fortran the status is an integer array, status(MPI_STATUS_SIZE), and the source rank is an element of this array which you can grab via status(MPI_SOURCE)

接收任何人的数据

- 请记住，一个参与者可能会（不可预测地）从任何其他参与者接收数据

  – 因此，通常使用 MPI_ANY_SOURCE 代替接收方的显式进程 ID – 可以查看 MPI 状态来确定发送方的 pid

```
MPI_Request请求;
MPI_Status状态;

MPI_Irecv（&消息，3，MPI_INT，
          MPI_ANY_SOURCE，...，
&请求）MPI_Wait（&请求，&状
态）；

int 源=状态.MPI_SOURCE;
```

```
MPI_Status 状态;
int 未偿付;

MPI_Iprobe（MPI_ANY_SOURCE，...，
优秀，&status）如果（优秀）{int source =
status.MPI_SOURCE;

    ......
}
```

在 Fortran 中，状态是一个整数数组，status(MPI_STATUS_SIZE)，源等级是此数组的一个元素，您可以通过 status(MPI_SOURCE) 获取它

Creation and Destruction of Actors (1)

epcc

- One of the requirements of an Actor is that it can create other actors.
- Even if you're doing a 1:1 mapping of actors to UEs you often want to avoid creating new MPI processes when creating a new actor.

- Solution: A process pool
  - At the start of the program, launch more processes than you'll ever have actors
  - Ensure that the program never creates more actors than this limit

Actor 的创建与销毁（1）

- Actor 的要求之一是它可以创建其他 Actor。

- 即使你正在将 Actor 和 UE 进行 1：1 的映射，你
  在创建新参与者时，通常希望避免创建新的 MPI 进程。

- 解决方案：进程池
  – 在程序启动时，启动比你以往任何时候更多的进程
  有演员——确保程序不会创建超过此限制的演员数量

Creation and Destruction of Actors (2)

- The problem with the solution:
  - How do the actors know if there are processes left in the pool?

- The solution:
  - Use a master process to manage new actors
  - Have a special master process with its own actor whose job it is to manage the process pool
  - When an actor wants to create a new actor, it sends a message to the master with the required information, and it's the master's job to assign an MPI process from the process pool to the new actor
    - You effectively use a master-worker pattern with the worker's task being: become an actor, and keep going through your event loop until you die

Actor 的创建与销毁（2）

- 解决方案存在的问题：
  - 参与者如何知道池中是否还有进程?

- 解决方案：
  - 使用主进程来管理新的 Actor – 有一个特殊的主进程，它有自己的 Actor，负责管理进程池 – 当 Actor 想要创建新 Actor 时，它会向

向船长提供所需的信息，船长的工作是
从进程池中分配一个 MPI 进程给新的参与者

- 有效地利用主从模式，完成工人的任务
  存在：成为一名演员，并不断经历事件循环，直到你死去

Process Pool **Pseudocode**

epcc

```
// Warning - This is pseudocode designed to illustrate an idea. Not all detail is included.

#define ACTORSTART 10          //Sent from the master to start an actor (from process pool)
#define SHUTDOWN 20            //Sent from the master to stop a worker
#define NEWACTORREQUEST 30     //Sent to the master to request a new actor
#define ACTORDIDDIE 40         //Sent by an actor who has died

int main(){
     int* busy;                //An array to keep track of which processes are in use
     MPI_Init()
     MPI_Comm_rank(myrank)
     MPI_Comm_size(nprocs)
     MPI_Buffer_attach(buffersize)      // Pick quite a large buffer size if in doubt
```

进程池伪代码

```
// 警告 - 这是用于说明想法的伪代码。并未包含所有细节。


#define ACTORSTART 10 //由主进程发送以启动一个参与者（来自进程池）
#define SHUTDOWN 20 //由主服务器发送以停止工作服务器
#define NEWACTORREQUEST 30 //发送给主进程以请求新的 Actor
#define ACTORDIDDIE 40 //由已去世的演员发送


int 主要(){
    int* busy; //一个数组，用于跟踪正在使用的进程 MPI_Init() MPI_Comm_rank(myrank) MPI_Comm_size(nprocs)
    MPI_Buffer_attach(buffersize) // 如果有疑问，请选择相当大的缓冲区大小
```

```
if(myrank == 0){                                          ! Master Process
      busy=malloc(nprocs*sizeof(int))
      for each proc{ busy[proc]=FALSE; }
      for (actor = 1; actor <= numactors; actor++){ // Start all the initial actors
           busy[actor]=TRUE;
           sendbuffer[0]=ACTORSTART;
           sendbuffer[1]=actortype;
           sendbuffer[2]=startdata;
           MPI_Bsend(sendbuffer, 3, MPI_INTEGER, actor,…);
      }
      do while !(programstop){
         MPI_Recv(event,3,MPI_INTEGER, MPI_ANY_SOURCE,status)
         sender=status.MPI_SOURCE
         if (event[0]==NEWACTORREQUEST){
                actor=findFreeActor(busy) // findFreeActor calls MPI_abort()if there is not one
                busy[actor]=TRUE;
                sendbuffer[0]=ACTORSTART;
                sendbuffer[1]=actortype;
                sendbuffer[2]=startdata;
                MPI_Bsend(sendbuffer, 3, MPI_INTEGER, actor,…)
         } else if (event[0]==ACTORDIDDIE){
                busy[sender]=FALSE;
         }
      } // end do while
      for each proc{
         sendbuffer[0]=SHUTDOWN;
         MPI_Bsend(sendbuffer,3, MPI_INTEGER, proc,…);
      }
} // end if(master)
```

```
if(myrank == 0){ !主进程
        busy=malloc(nprocs*sizeof(int)) for each proc{ busy[proc]=FALSE; } for (actor = 1; actor <= numactors;
        actor++){//启动所有初始actor


        busy[actor]=TRUE; sendbuffer[0]=ACTORSTART;
        sendbuffer[1]=actortype; sendbuffer[2]=startdata;
        MPI_Bsend(sendbuffer, 3, MPI_INTEGER, actor,…); }




        当! （programstop）时执行{MPI_Recv（event，3，MPI_INTEGER，
        MPI_ANY_SOURCE，status）发送方=status.MPI_SOURCE，如果（event [0] ==
        NEWACTORREQUEST）{

                actor=findFreeActor(busy) //如果没有，findFreeActor 会调用 MPI_abort() busy[actor]=TRUE;
                sendbuffer[0]=ACTORSTART; sendbuffer[1]=actortype; sendbuffer[2]=startdata; MPI_Bsend(sendbuffer, 3,
                MPI_INTEGER, actor,…)




            }否则，如果（事件[0]==ACTORDIDDIE）{
                忙[发送者] = FALSE;
            }
        } // 结束执行
        对于每个过程{
        sendbuffer[0]=SHUTDOWN;
            MPI_Bsend(发送缓冲区，3，MPI_INTEGER，proc，…);
} } // 如果（主）则
结束
```

```
else{                                                    //worker
   do while (die==FALSE){
     MPI_Recv(event, 3, MPI_INTEGER, …)
     if (event[0]==ACTORSTART){
               switch(event[1]){
               case(ACTORTYPE1):
                         actorType1EventLoop(event[2]);
                         break;
               case(ACTORTYPE2):
                         actorType2EventLoop(event[2]);
                         break;
               …
               }
     } else if (event[0]==SHUTDOWN){
               die=TRUE;
     }
   } // end do
   MPI_Finalize()

} // end else
```

```
else{//工人执行 while (die==FALSE){

        MPI_Recv (事件, 3, MPI_INTEGER, ...)
        如果（事件[0]==ACTORSTART）{
                    开关（事件[1]）{
                    案例（ACTORTYPE1）：
                                actorType1EventLoop(事件[2]); 中断;

                    案例（ACTORTYPE2）：
                                actorType2EventLoop（事件[2]）; 中断;

                    ...
        } } 否则，如果（事件
        [0]==SHUTDOWN）{
                    死亡=真;
        }
    } // 结束
    MPI_Finalize ()

} // 否则结束
```

Conclusions

- We have talked about the Actor Pattern, similar to event based co-ordination but with some differences
  - Actors can create other actors and die
  - No need for external events
  - Actors can perform work not driven by messages
- A useful pattern which has the potential to be very important in the future
  - The loose synchronisation might be crucial for extremely large core counts
- MPI isn't a perfect fit for the implementation, but can be used
  - Dynamic creation/destruction of actors is tricky
  - An ideal candidate for use as a framework.

结论

- 我们讨论了 Actor 模式，它类似于基于事件的协调，但有一些区别
  - Actor 可以创建其他 Actor，然后消亡 – 不需要外部事件 – Actor 可以执行不受消息驱动的工作
- 这是一个有用的模式，在未来可能会非常重要
  - 对于非常大的核心数，松散同步可能至关重要
- MPI 并不完美适合实现，但可以使用
  - 动态创建/销毁参与者很棘手 – 用作框架的理想候选者。