# Privacy-Preserving Analytics in and out of the Clouds

Jon Crowcroft,
http://www.cl.cam.ac.uk/~jac22

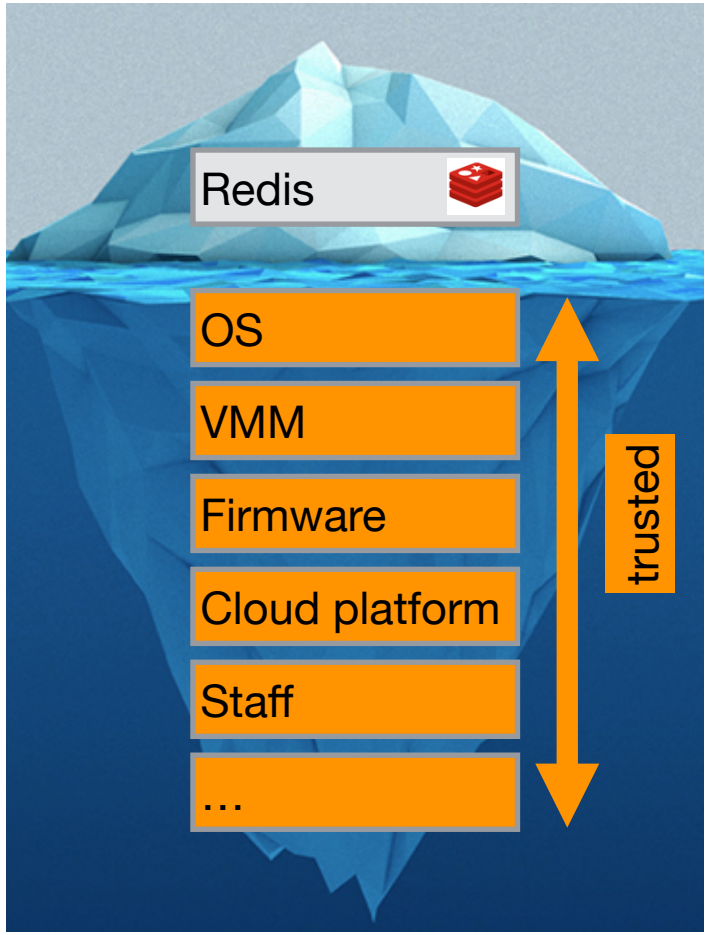
This work is funded in part by the EPSRC Databox (EP/N028260/1),
NaaS (EP/K031724/2) and Contrive (EP/N028422/1) and Turing/Maru (EP/N510129/1) projects.

# 1. Trust Issues: Provider Perspective

Cloud provider does not trust users

Use virtual machines to isolate users from each other and the host

VMs only provide one way protection

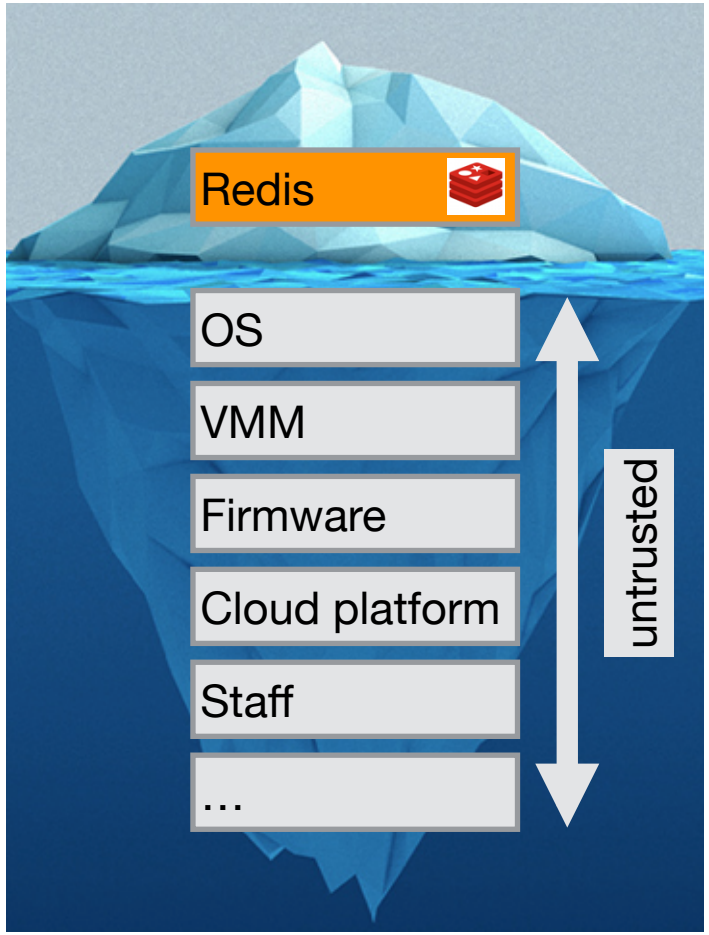


# Trust Issues: User Perspective

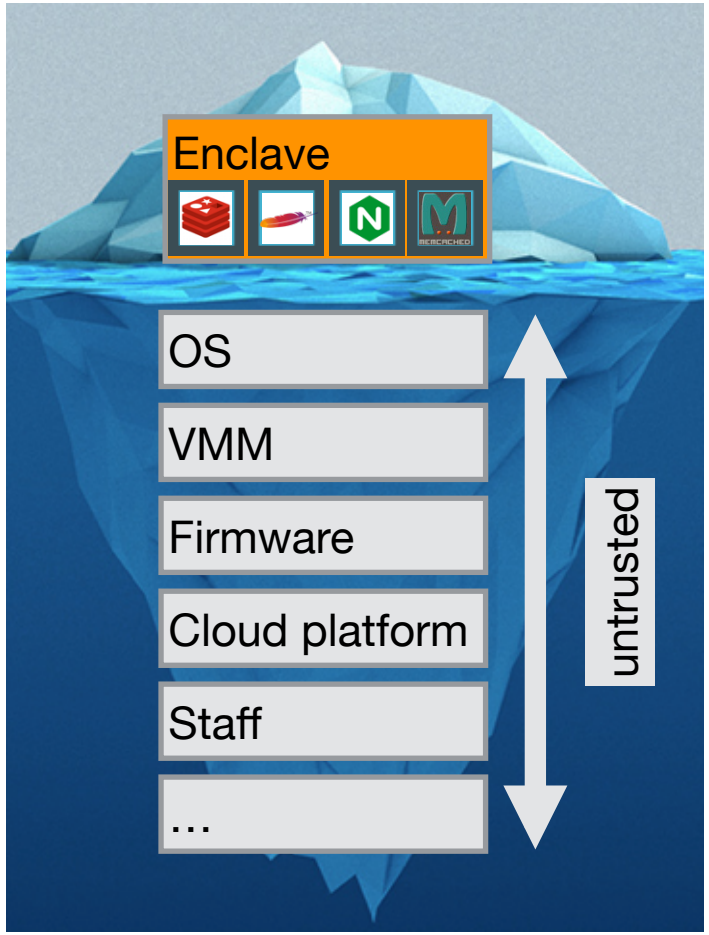
Users trust their applications

Users must implicitly trust cloud provider

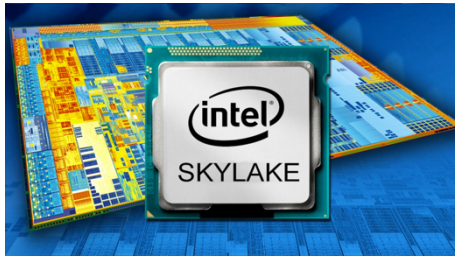
Existing applications implicitly assume trusted operating system



# Trusted Execution Support with Intel SGX



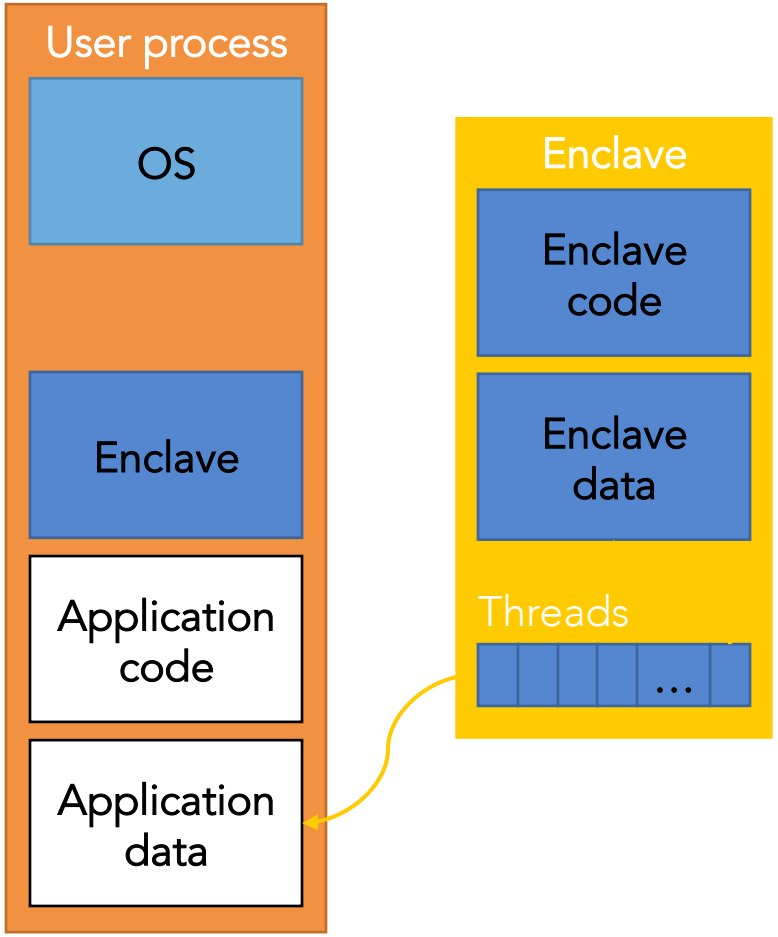

- Users create HW-enforced trusted environment (enclave)
- Supports unprivileged user code
- Protects against strong attacker model
- Remote attestation
- Available on commodity CPUs



# Trusted Execution Environments

Trusted execution environment (TEE) in process

- Own code & data
- Controlled entry points
- Provides confidentiality & integrity
- Supports multiple threads
- Full access to application memory



# Intel Software Guard Extensions (SGX)

Extension of Instruction Set Architecture (ISA) in recent Intel CPUs

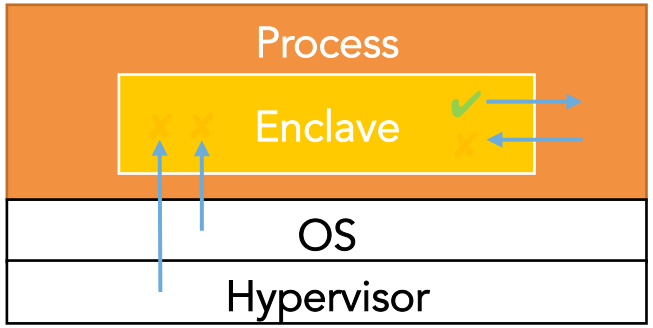
- Skylake (2015), Kaby lake (2016)

Protects confidentiality and integrity of code & data in untrusted environments

- Platform owner considered malicious
- Only CPU chip and isolated region trusted

# SGX Enclaves

SGX introduces notion of **enclave**
- Isolated memory region for code & data
- New CPU instructions to manipulate enclaves and new enclave execution mode




Enclave memory **encrypted** and **integrity-protected** by hardware
- Memory encryption engine (MEE)
- No plaintext secrets in main memory

Enclave memory can be accessed only by enclave code
- Protection from privileged code (OS, hypervisor)

Application has ability to defend secrets
- Attack surface reduced to just enclaves and CPU
- Compromised software cannot steal application secrets

# SGX SDK Code Sample

SGX application: untrusted code

```
char request_buf[BUFFER_SIZE];
char response_buf[BUFFER_SIZE];

int main()
{
  ...
  while(1)
  {
    receive(request_buf);
    ret = EENTER(request_buf, response_buf);
    if (ret < 0)
      fprintf(stderr, "Corrupted message\n");
    else
      send(response_buf);
  }
  ...
}
```

Enclave: trusted code

```
char input_buf[BUFFER_SIZE];
char output_buf[BUFFER_SIZE];

int process_request(char *in, char *out)
{
  copy_msg(in, input_buf);
  if(verify_MAC(input_buf))
  {
    decrypt_msg(input_buf);
    process_msg(input_buf, output_buf);
    encrypt_msg(output_buf);
    copy_msg(output_buf, out);
    EEXIT(0);
  } else
    EEXIT(-1);
}
```

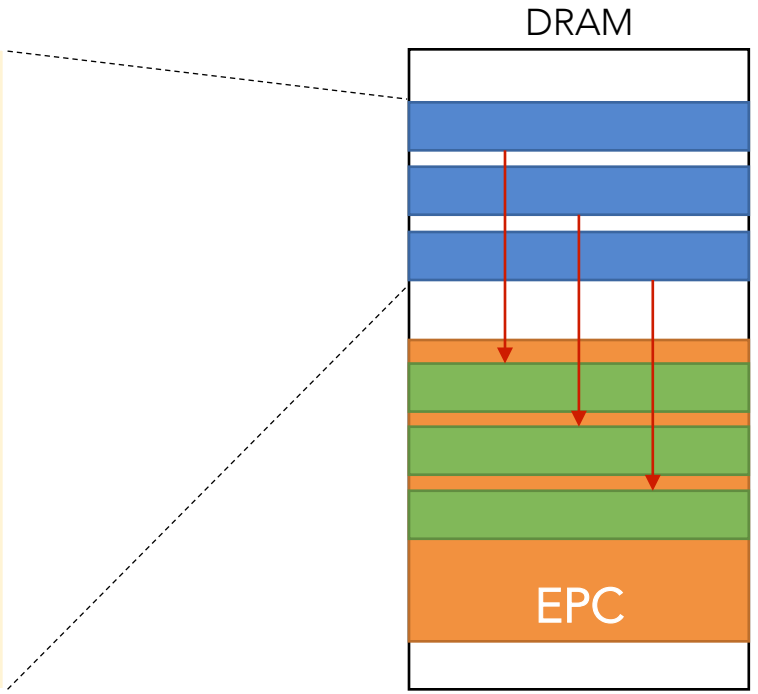
Server:

- Receives encrypted requests
- Processes them in enclave
- Sends encrypted responses

# SGX Enclave Construction

```
1 { char input_buf[BUFFER_SIZE];
2 { char output_buf[BUFFER_SIZE];

  int process_request(char *in, char *out)
  {
    copy_msg(in, input_buf);
    if(verify_MAC(input_buf))
    {
      decrypt_msg(input_buf);
3     process_msg(input_buf, output_buf);
      encrypt_msg(output_buf);
      copy_msg(output_buf, out);
      EEXIT(0);
    } else
      EEXIT(-1);
  }
```




Enclave populated using special instruction (EADD)

- Contents initially in untrusted memory
- Copied into EPC in 4KB pages

Both data & code copied before execution commences in enclave

# SGX Support for Spark

**SGX-Spark**

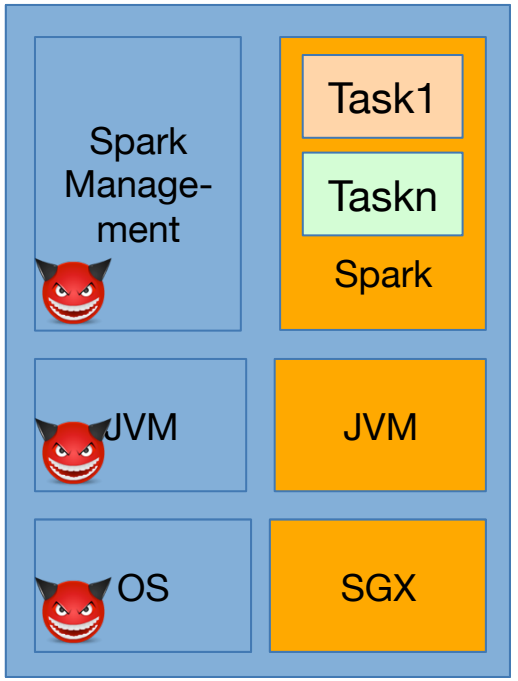
- Protect data processing from infrastructure provider
- Protect confidentiality & integrity of existing jobs
- No modifications for end users
- Acceptable performance overhead

Idea:
Execute only sensitive parts of Spark inside enclave

- Code that accesses/processes sensitive data

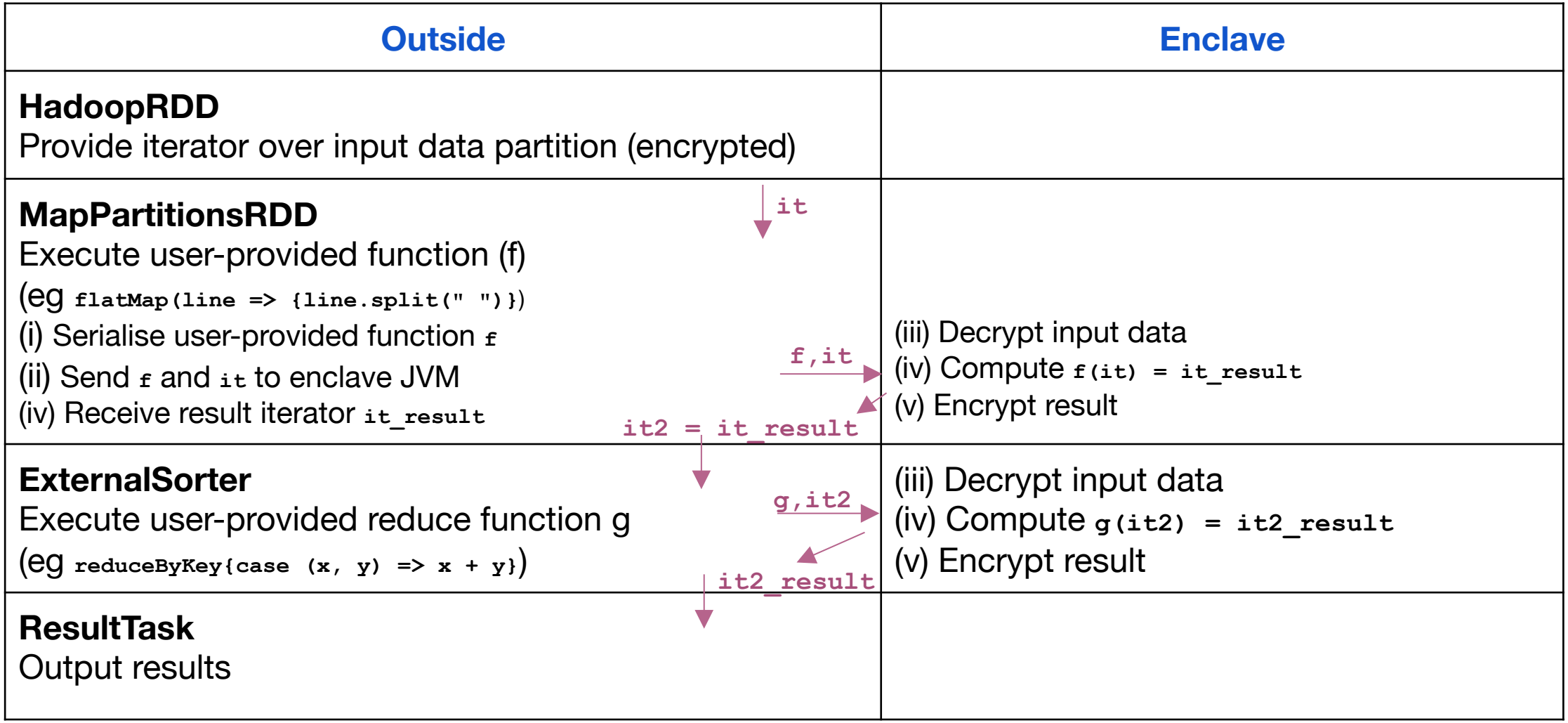
Code outside of enclave only accesses encrypted data

- Partition Spark
- Run two collaborating JVMs, inside enclave and outside of enclave

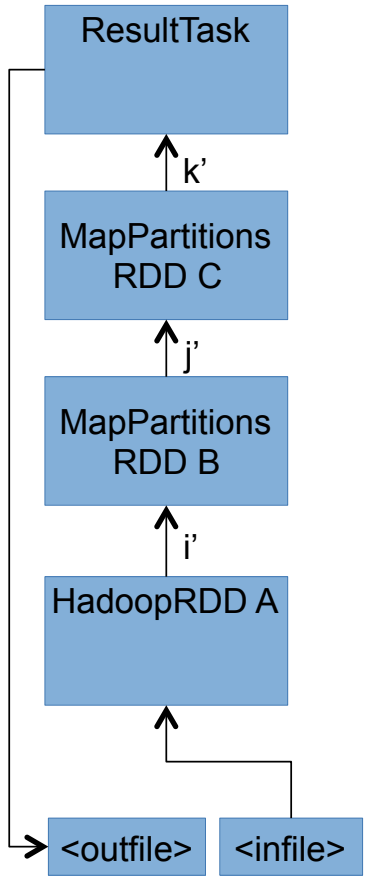


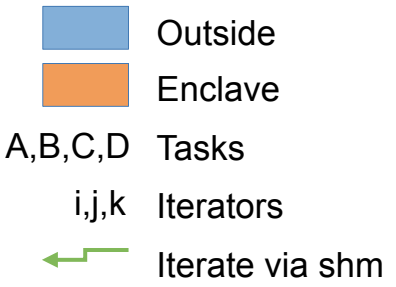
# Partitioning Spark

Goal: Move minimal amount of Spark code to enclave

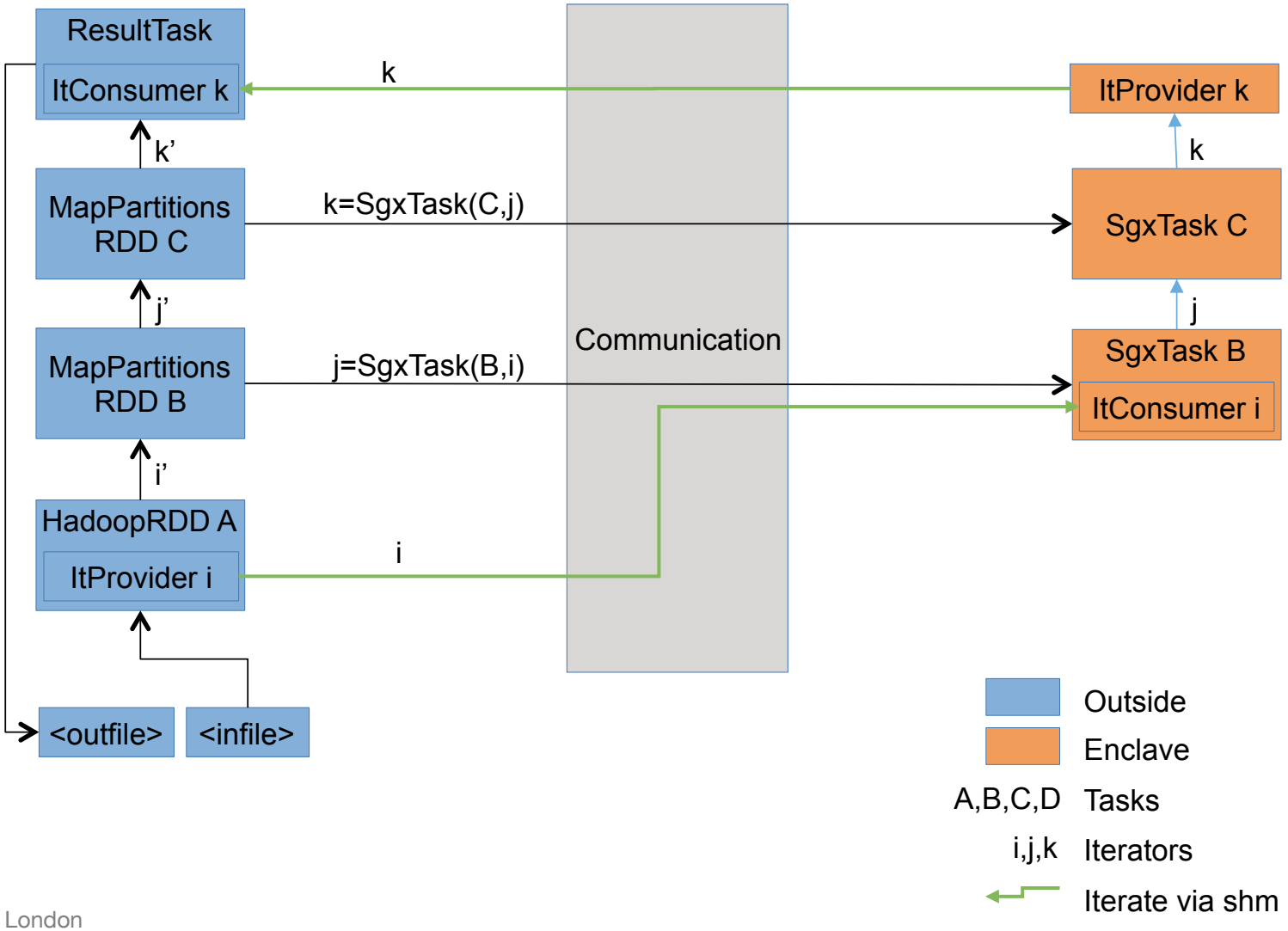


# Partitioning Spark



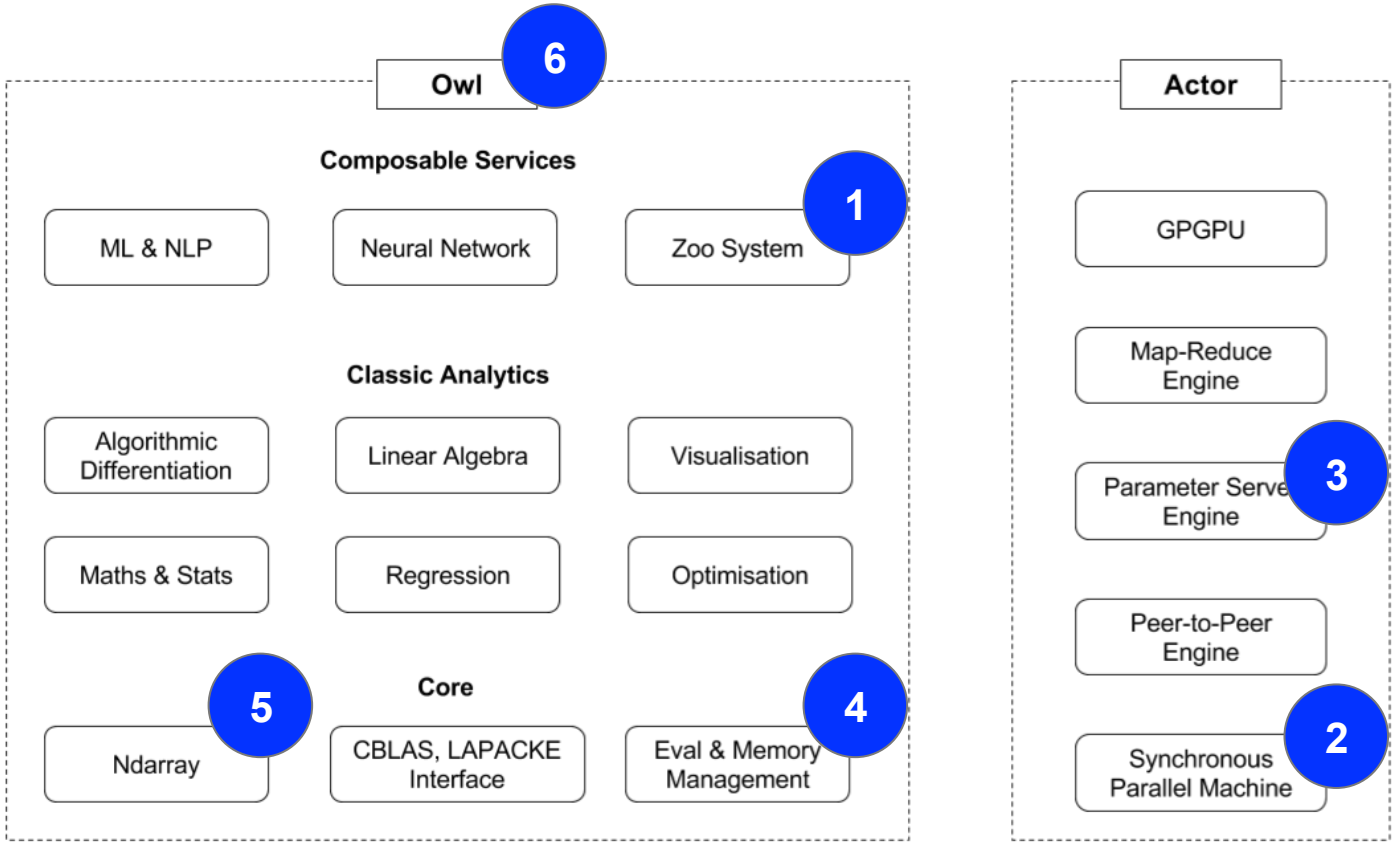
# Partitioning Spark



# 2. Owl Platform – distributed&parallel ML

- An experimental and above all scientific computing system.
- Designed in functional programming paradigm.
- Goal: as concise as Python yet as fast as C, and safe.
- A comprehensive set of classic numerical functions.
- A fundamental tooling for modern data analytics (ML & DNN).
- Native support for algorithmic differentiation, distributed & parallel computing, and GPGPU computing.

# Research & Owl Architecture




1. Jiaxin Zhao - *Composable Analytical Services using Session Types for Distributed Personal Data*

2. Ben Catterall - *Probabilistic Synchronous Parallel - A New Barrier Control Method for Distributed Machine Learning*

3. D.S.R Royson - *Optimising Adaptive Learning in Distributed Machine Learning*

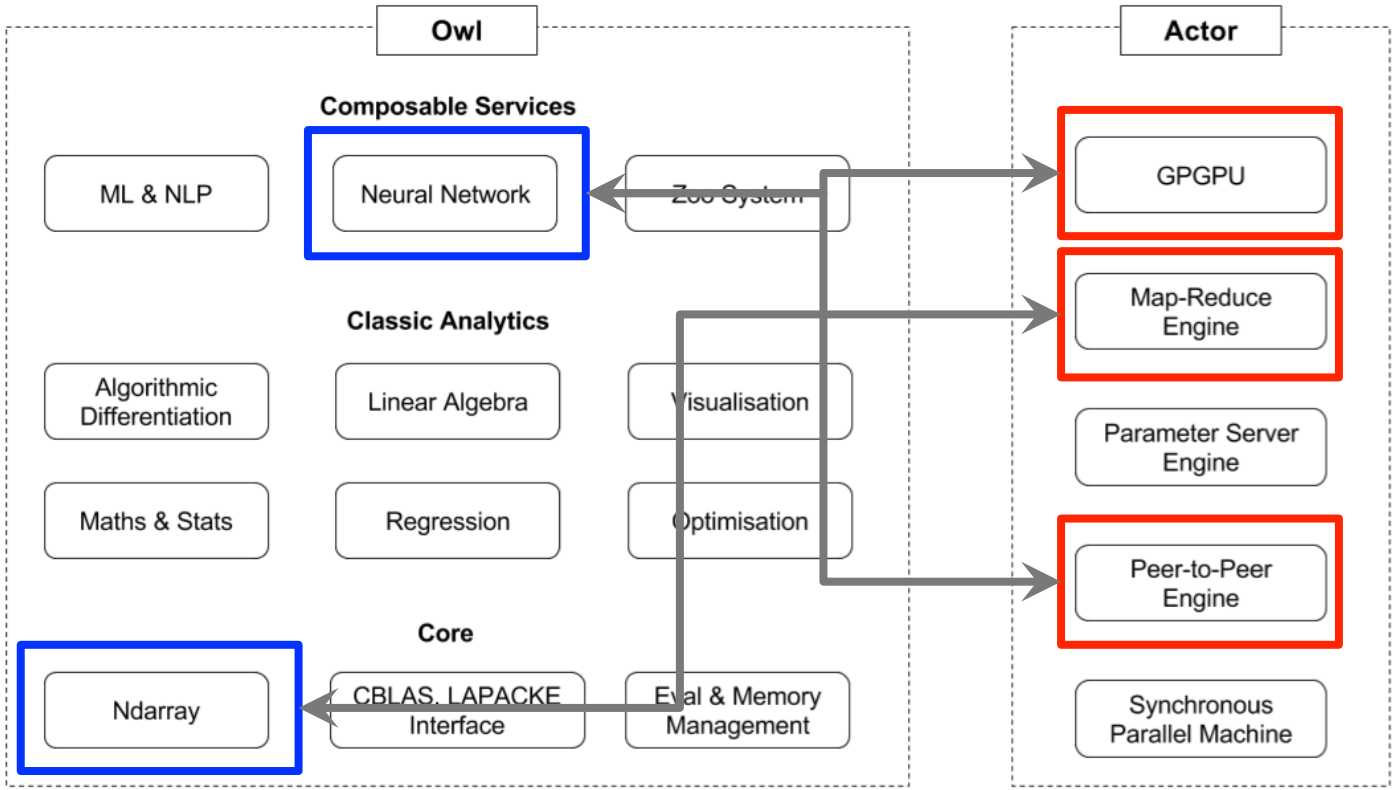
4. Dhruv Makwana - *Memory Management using Linear Types for High-Performance GPGPU Numerical Computing*

5. Tudor Tiplea - *Deploying Browser-based Data Analytics at Network Edge*

6. Liang Wang - *Owl: A General-Purpose Numerical Library in OCaml*
Presented in ICFP'17 OCaml meeting, tutorial in CUFP'17.

UNIVERSITY OF CAMBRIDGE

# Parallel and Distributed Computing




Parallel and distributed computing is achieved by composing the different data structures in Owl's core library with specific engines in Actor system.

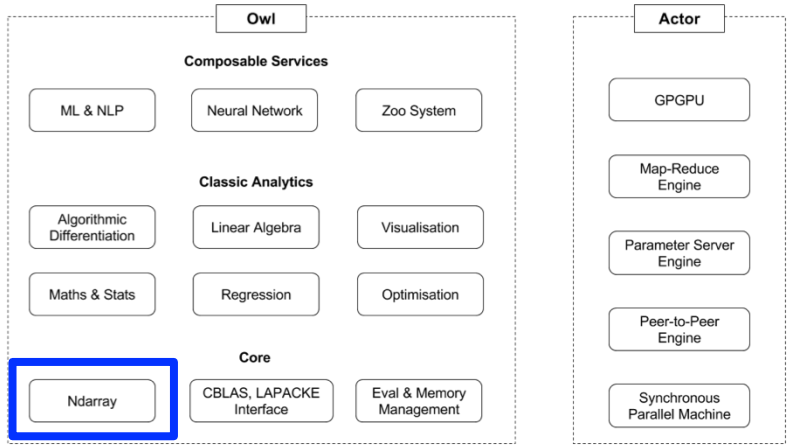
UNIVERSITY OF CAMBRIDGE

# Owl + Actor : Ndarray Example

A `map` function on local ndarray `x` in Owl looks like this

```
Dense.Ndarray.S.map sin x
```

How to implement a distributed `map` on distributed ndarray?



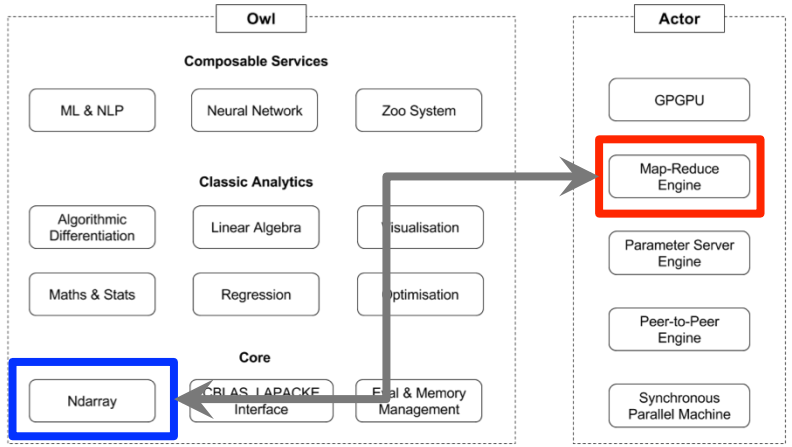
# Owl + Actor : Ndarray Example

Like playing LEGO, we plug Ndarray into Distribution Engine to make a distributed Ndarray.

```
module M = Owl.Parallel.Make (Dense.Ndarray.S) (Actor.Mapre)
```

Composed by a functor in `Owl_parallel` module, which connects two systems and hides details.

```
M.map sin x
```




UNIVERSITY OF CAMBRIDGE

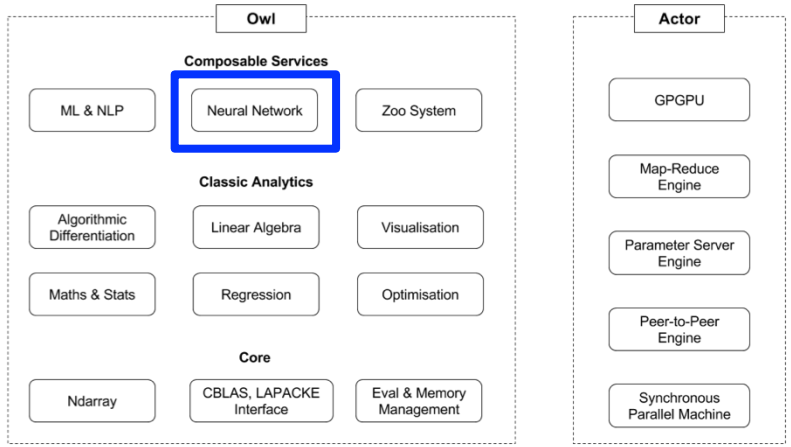
# Owl + Actor : Neural Network Example

Similarly, this also applies to more advanced and complicated data structures such as neural network. This is how we define a NN in Owl:

```
let network =
  input [|28;28;1|]
  |> lambda (fun x -> Maths.(x / F 256.))
  |> conv2d [|5;5;1;32|] [|1;1|] ~act_typ:Activation.Relu
  |> max_pool2d [|2;2|] [|2;2|]
  |> dropout 0.1
  |> fully_connected 1024 ~act_typ:Activation.Relu
  |> linear 10 ~act_typ:Activation.Softmax
  |> get_network
```

Then we can perform training locally as below

```
Owl.Neural.S.Graph.train network
```



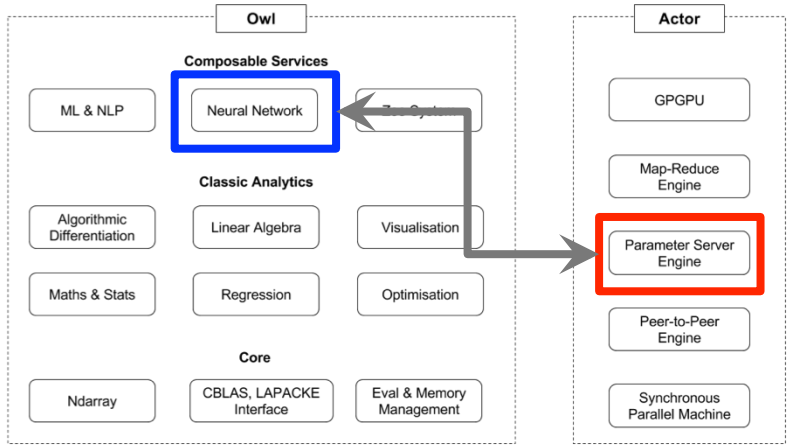
# Owl + Actor : Neural Network Example

To enable the parallel training on a computer cluster, we can combine `Graph` with `Param` engine.

```
module M = Owl.Parallel.Make (Owl.Neural.S.Graph) (Actor.Param)
```

We only write code once, Owl's functor stack generates both sequential and parallel version for us!

```
M.train network
```

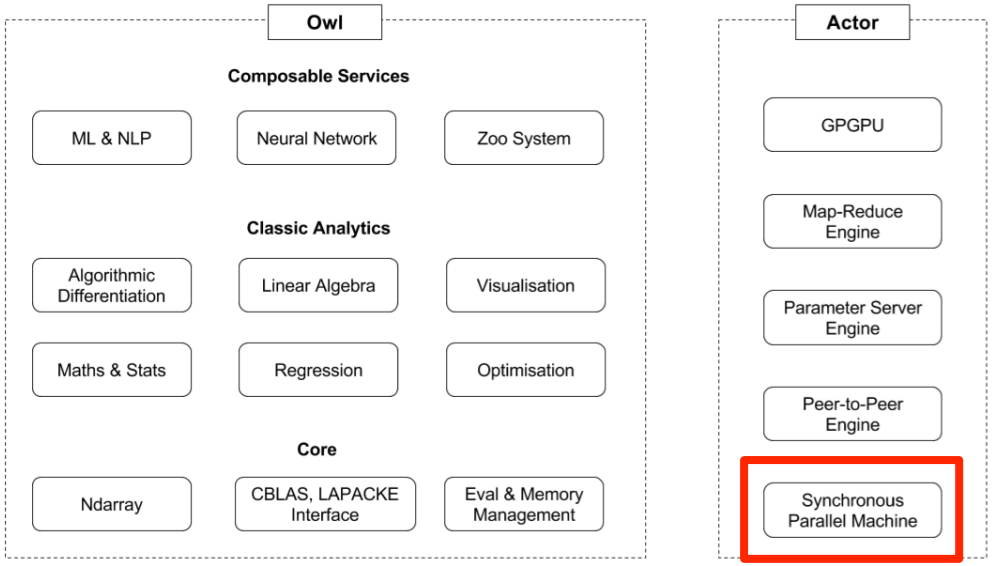



UNIVERSITY OF CAMBRIDGE

# Key to Scalability

Actor implements three engines, maybe more in future.

All reply on a module called Synchronous Parallel which handles synchronisation.




Corner stone of large scale distributed learning :)

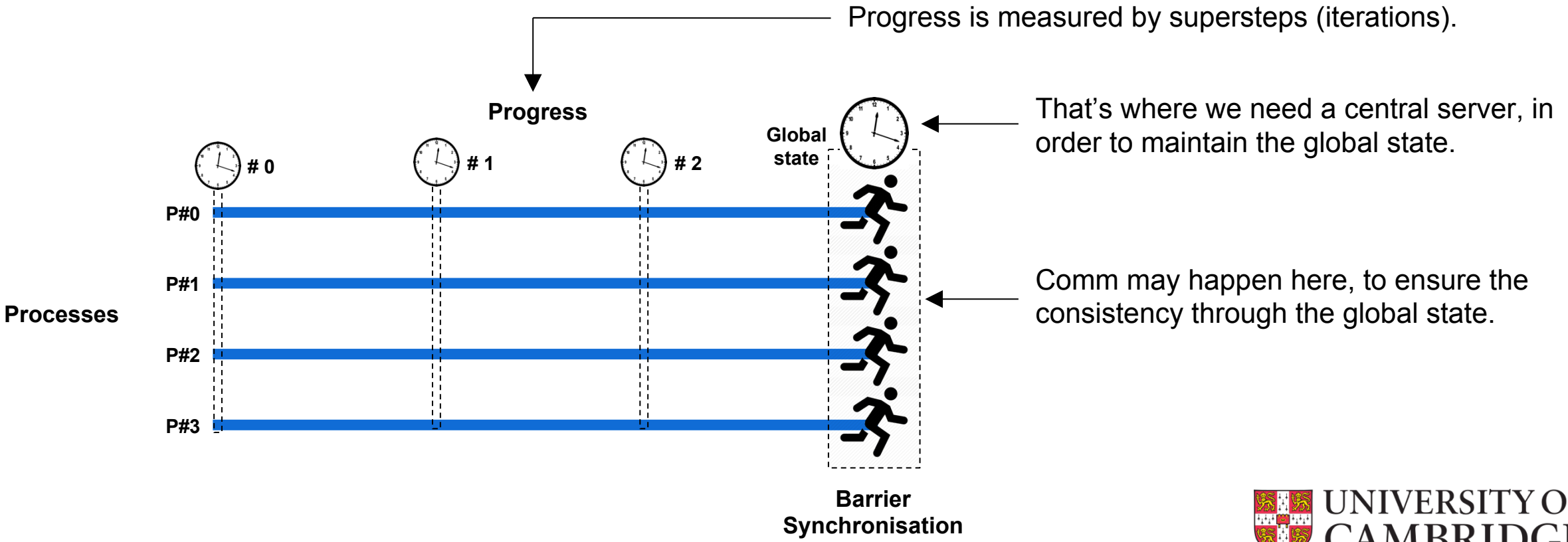
UNIVERSITY OF CAMBRIDGE

# Synchronous Parallel Machine

- An abstract computer for designing parallel algorithms.
- Three components:
  - A local processor equipped with fast memory;
  - A network that routes messages between computers;
  - A (hw/sw) component to synchronise all computers;
- Powerful model for designing and programming parallel systems, building block of Apache Hadoop, Spark, Hama, etc.

# Barrier Synchronisation

The third component barrier synchronisation, is the core of SPM. It's all about how to coordinate computation on different computers to achieve certain consistency.



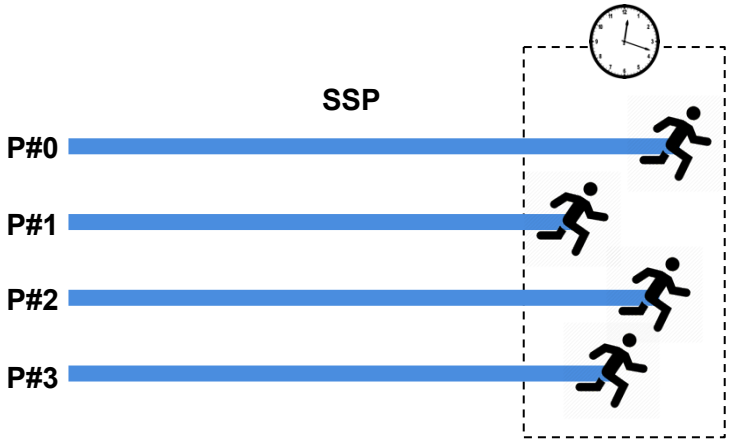
# Consistency Is Not Free Lunch

- Real world iterative learning algorithm aims fast convergence.
- Convergence rate decreases if iteration rate is slow or updates are noisy.
- Synchronisation can reduce the noise in updates (improved consistency).
- Tight synchronisation is sensitive to stragglers and has poor scalability.
- Tight synchronisation renders high communication cost in large systems.

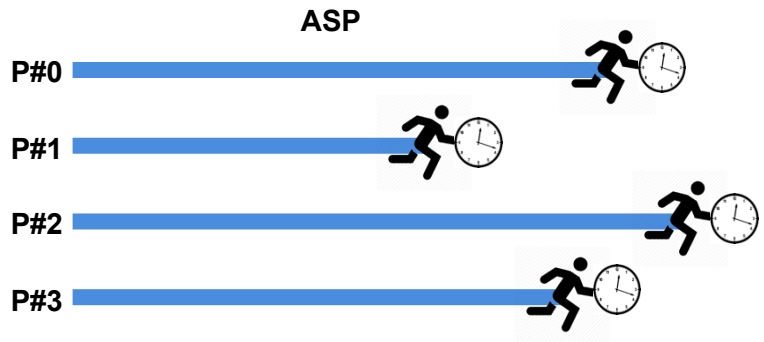
# Existing Models in Use

- Bulk synchronous parallel (BSP)
- Stale Synchronous parallel (SSP)
- Asynchronous parallel (ASP)


Hadoop, Spark, Parameter Server, Pregel, Owl+Actor ...




Parameter Server, Hogwild!, Cyclic Delay, Yahoo! LDA, Owl+Actor ...



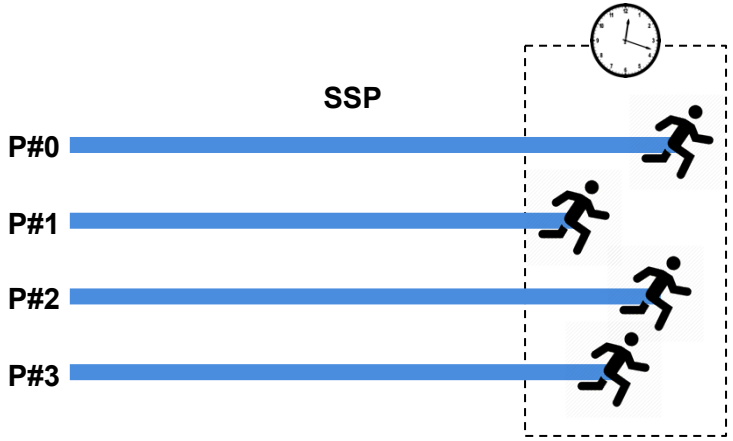

Parameter Server, Hogwild!, Yahoo! LDA, Owl+Actor ...

UNIVERSITY OF CAMBRIDGE

# A 10,000 Foot View

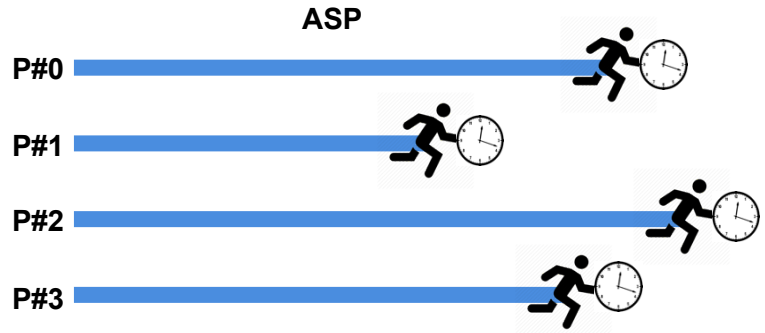

Most strict lockstep synchronisation; all the nodes are coordinated by a central server.

BSP is sensitive to stragglers so is very slow. But it is simple due to its deterministic nature, easy to write application on top of it.




SSP relaxes consistency by allowing difference in iteration rate. The difference is controlled by the bounded staleness.

SSP is supposed to mitigate the negative effects of stragglers. But the server still requires global state.

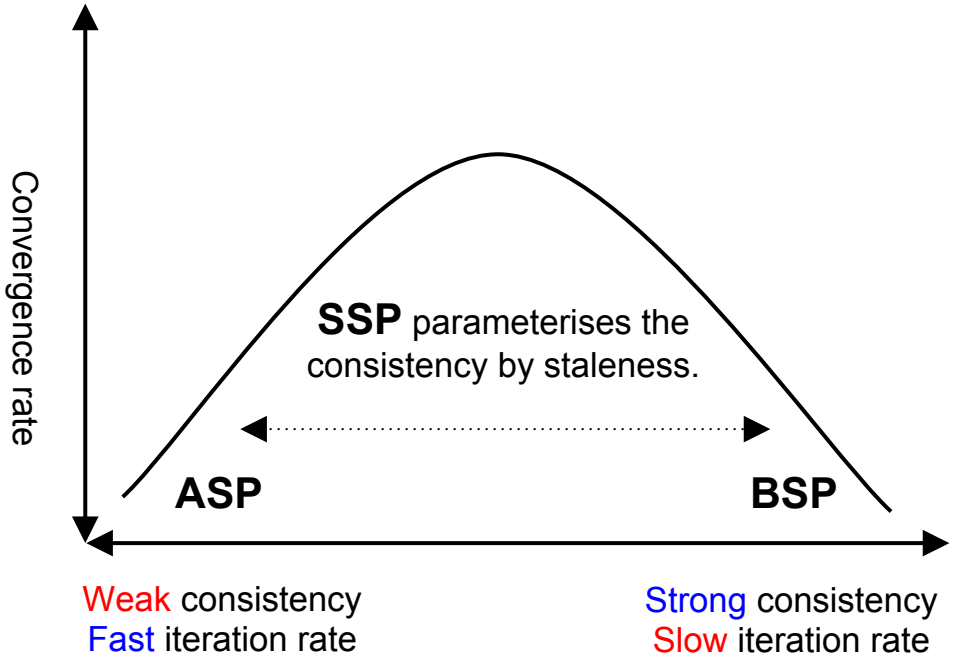



Least strict synchronisation, no communication among workers for barrier synchronisation all all.

Every computer can progress as fast as it can. It is fast and scalable, but often produces noisy updates. No theoretical guarantees on consistency and algorithm's convergence.

UNIVERSITY OF CAMBRIDGE

# Consistency vs. Iteration Rate




We must balance consistency vs. iteration rate , we can think in the following way to understand the trade-off:

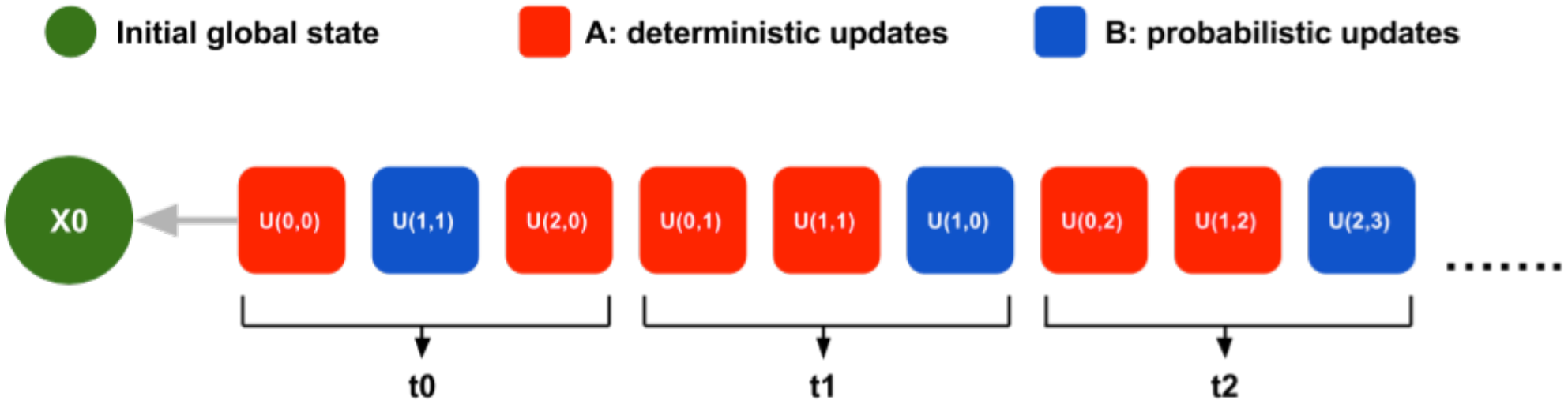
**Convergence $\propto$ Consistency Degree x Iteration Rate**

SSP aims to cover this spectrum between BSP and ASP by parameterising the staleness.

**Question**: Is SSP really a generalisation of of BSP and ASP?
BSP and SSP are both centralised whereas ASP are fully distributed. It feels like SSP has missed something in this spectrum. What is it?

UNIVERSITY OF CAMBRIDGE

# Analytical Model




$$\mathbf{x_0} \quad += \quad \left[ \sum_{\mathcal{A}} \mathbf{u_{p,t}} \right] \quad + \quad \left[ \sum_{\mathcal{B}} \mathbf{u_{p,t}} \right]$$

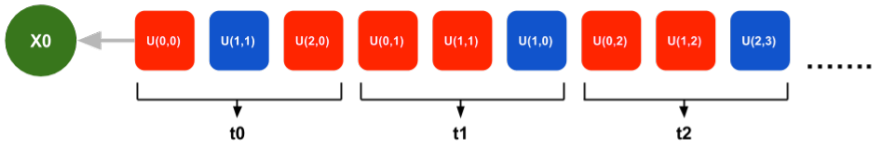
`u(p,t)` is `update(node id, timestamp)`, sum over all the nodes and clock ticks ...

The model is simple: a sequence of updates applying to an initial global state **x0**. The updates can be "noisy" hence are divided into two sets.

Although simple, it can model most iterative learning algorithms.

UNIVERSITY OF CAMBRIDGE

# Decompose Synchronous Parallel Machine




$$\text{BSP} \quad \mathbf{x_0} \; += \; \left[ \sum_{\mathcal{A}} \mathbf{u_{p,t}} \right]$$

$$\text{SSP} \quad \mathbf{x_0} \; += \; \left[ \sum_{\mathcal{A}} \mathbf{u_{p,t}} \right] + \left[ \sum_{\mathcal{B}} \mathbf{u_{p,t}} \right]$$

$$\text{ASP} \quad \mathbf{x_0} \; += \; \left[ \sum_{\mathcal{B}} \mathbf{u_{p,t}} \right]$$

Then we use the analytical model to express each synchronous parallel machine on the left.

The formulation reveals some very interesting structures from a system design perspective.

Let's decompose these machines.

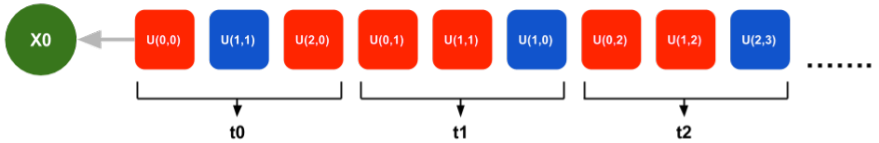
# Decompose Synchronous Parallel Machine

Consistency | Synchronisation

BSP $\quad \mathbf{x_0} \; += \left[ \sum_{\mathcal{A}} \mathbf{u_{p,t}} \right]$

SSP $\quad \mathbf{x_0} \; += \left[ \sum_{\mathcal{A}} \mathbf{u_{p,t}} \right] + \left[ \sum_{\mathcal{B}} \mathbf{u_{p,t}} \right]$

ASP $\quad \mathbf{x_0} \; += \left[ \sum_{\mathcal{B}} \mathbf{u_{p,t}} \right]$

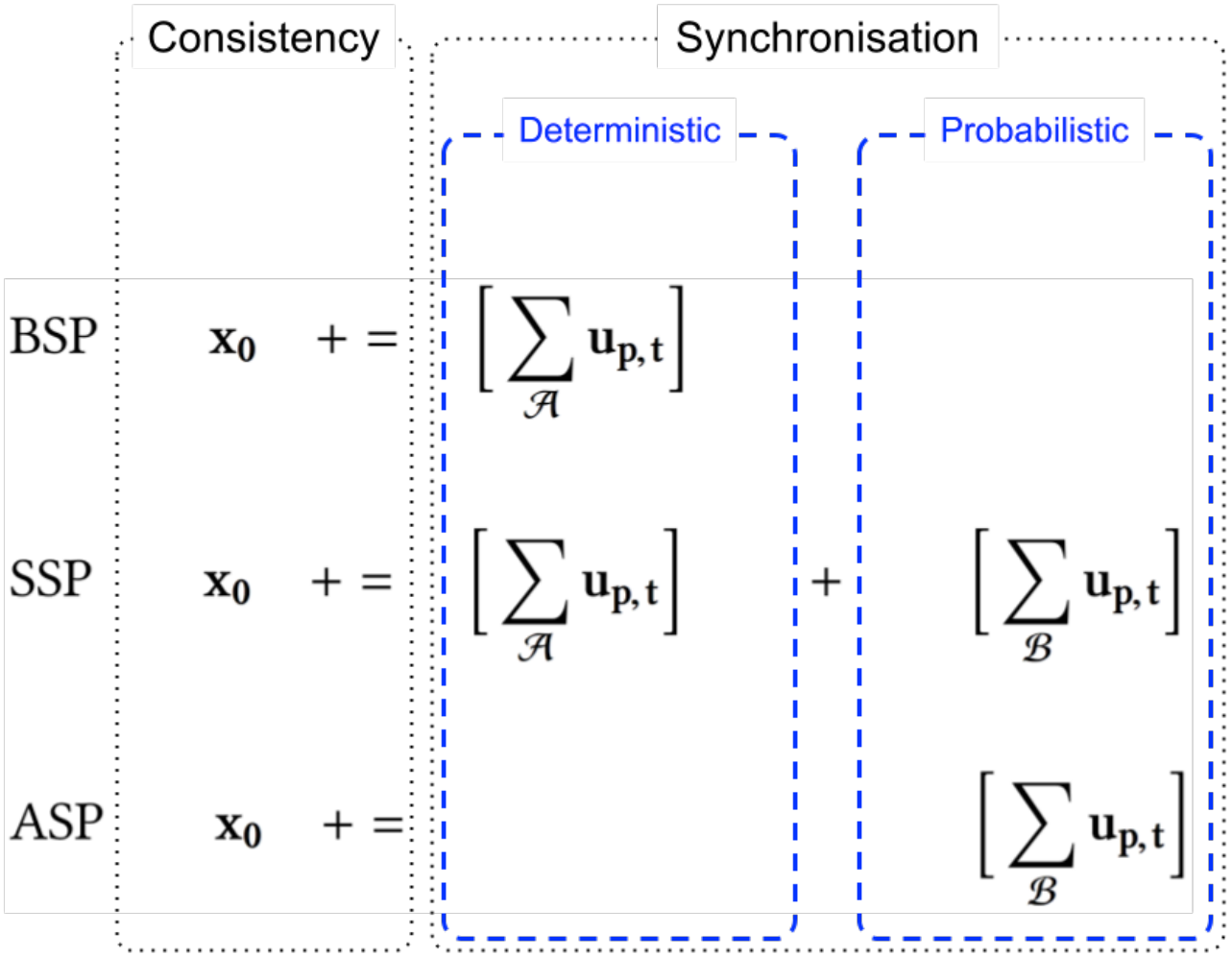


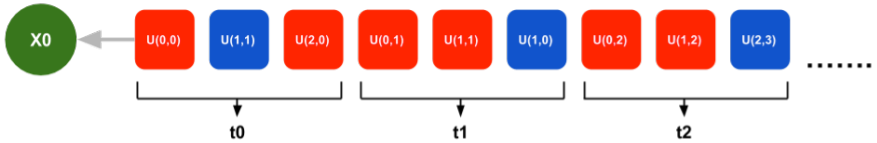

Left part deals with the Consistency. **+=** operator is the server logic about how to incorporate updates submitted to the central server into the global state.

Right part deals with synchronisation, computers either communicate to each other or contact the central server to coordinate their progress.

UNIVERSITY OF CAMBRIDGE

# Decompose Synchronous Parallel Machine



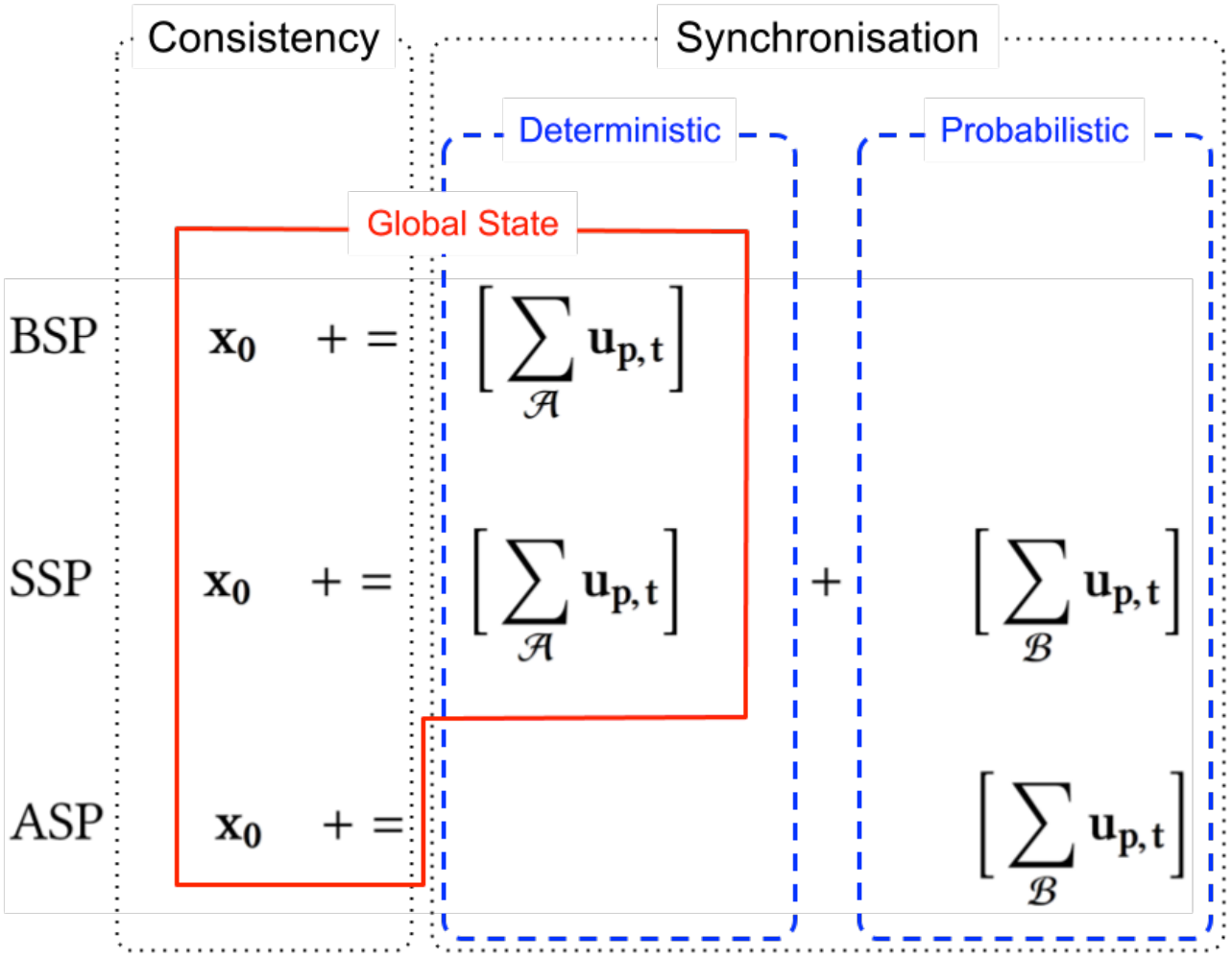

Inside the synchronisation part, the synchronous parallel machine processes two types of updates.

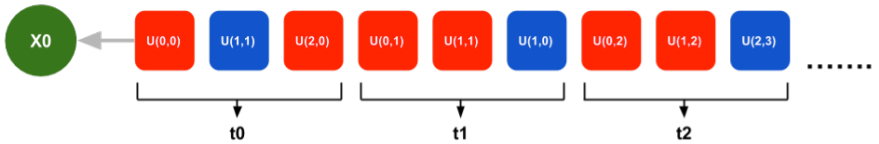
The deterministic one is those we always expect if everything goes well as in BSP.

The probabilistic one is those out of order updates due to packet loss, network delay, node failure, and etc.

UNIVERSITY OF CAMBRIDGE

# Decompose Synchronous Parallel Machine



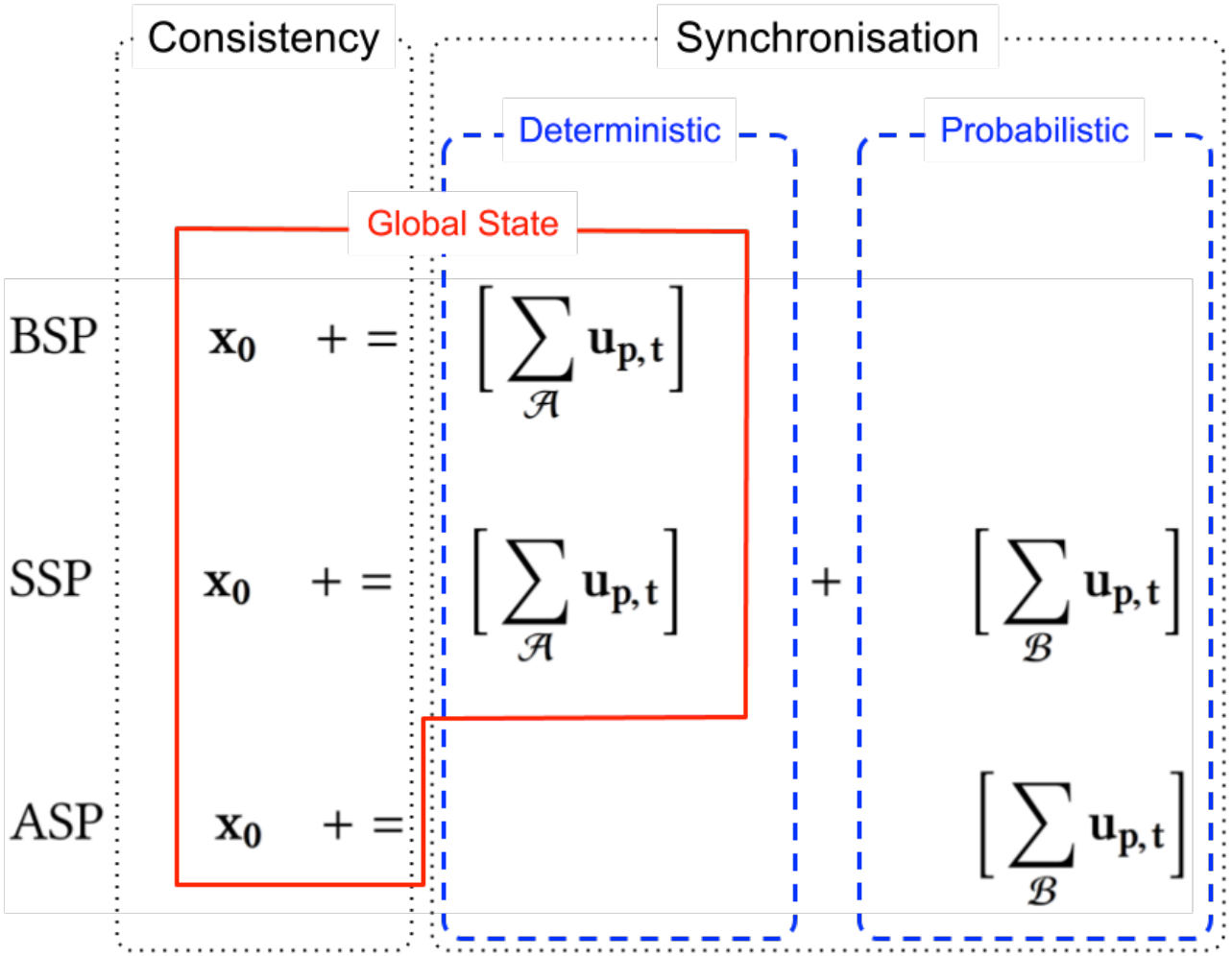

The global state is maintained by a logic central server. In a distributed system, this is often the bottleneck and single point of failure.

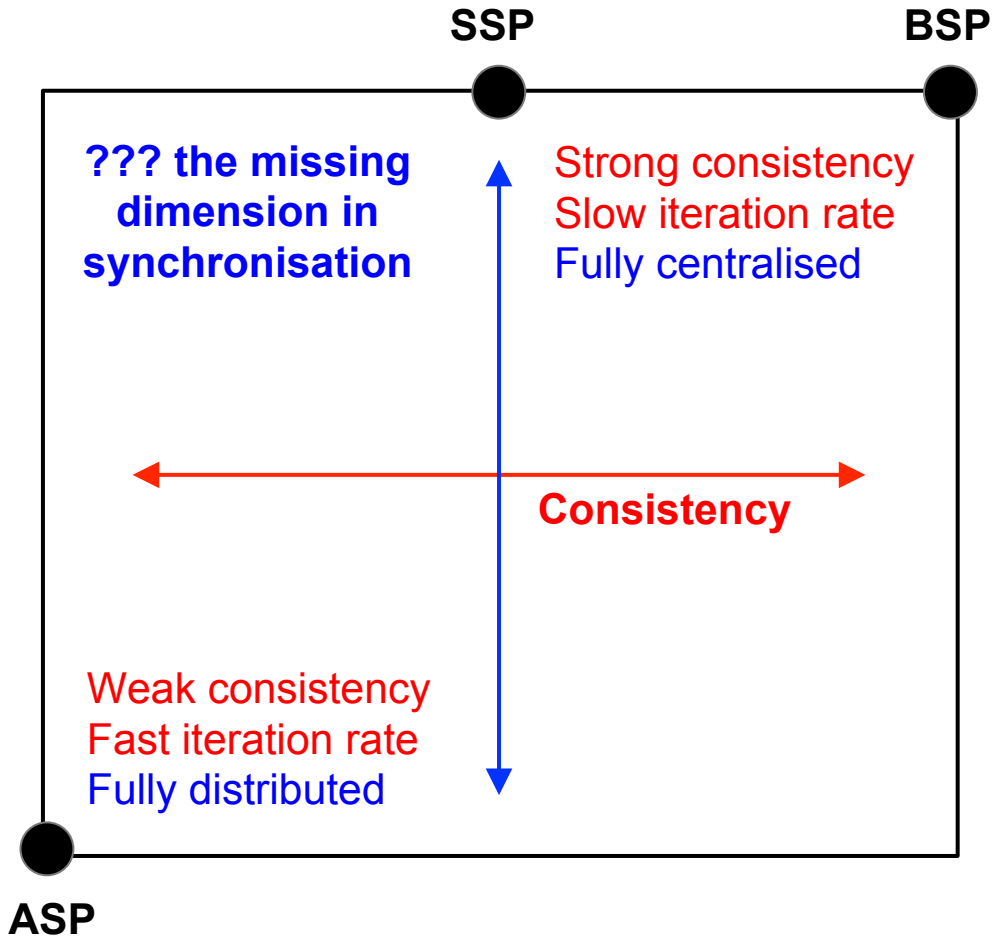
Moreover, note that the server couples the consistency with the synchronisation two parts, for BSP and SSP.

ASP avoids such coupling by giving up synchronisation, consistency completely.

UNIVERSITY OF CAMBRIDGE

# Missing Dimension in Design Space




UNIVERSITY OF CAMBRIDGE

# Key Insights from Decomposition

- Is it necessary to couple the consistency with synchronisation? No
- Is it necessary to give up synchronisation completely in order to decouple consistency and synchronisation? No
- Is it necessary to divide the updates into deterministic and probabilistic two parts? No
- Is SSP really a generalisation of BSP and ASP, then what is the missing dimension in the design space? Completeness

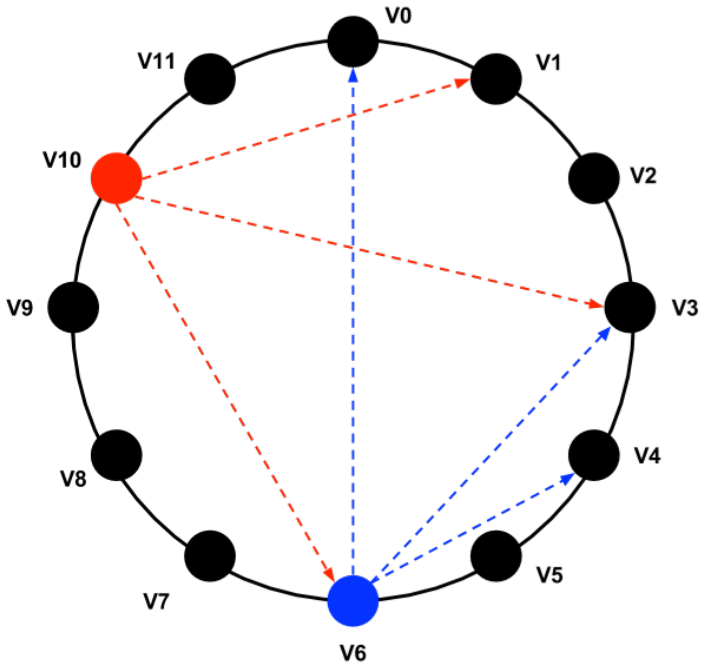
# Probabilistic Synchronous Parallel

- Core idea: combine both deterministic and probabilistic components, replace it with a sample distribution.
- Each computer synchronises with a small group of others and the consistency is only enforced within the group.
- The server decides how to incorporate the submitted updates.


No central server to coordinate them, instead each node synchronises within their groups. The consistency “propagates” by the possible overlapping of different groups.

# Sampling Primitive

- How to implement PSP atop of current data analytics frameworks?
  - Quite straightforward, add a new primitive - `sample`
- How to guarantee the random sampling?
  - Organise the nodes into a structural overlay, e.g. DHT

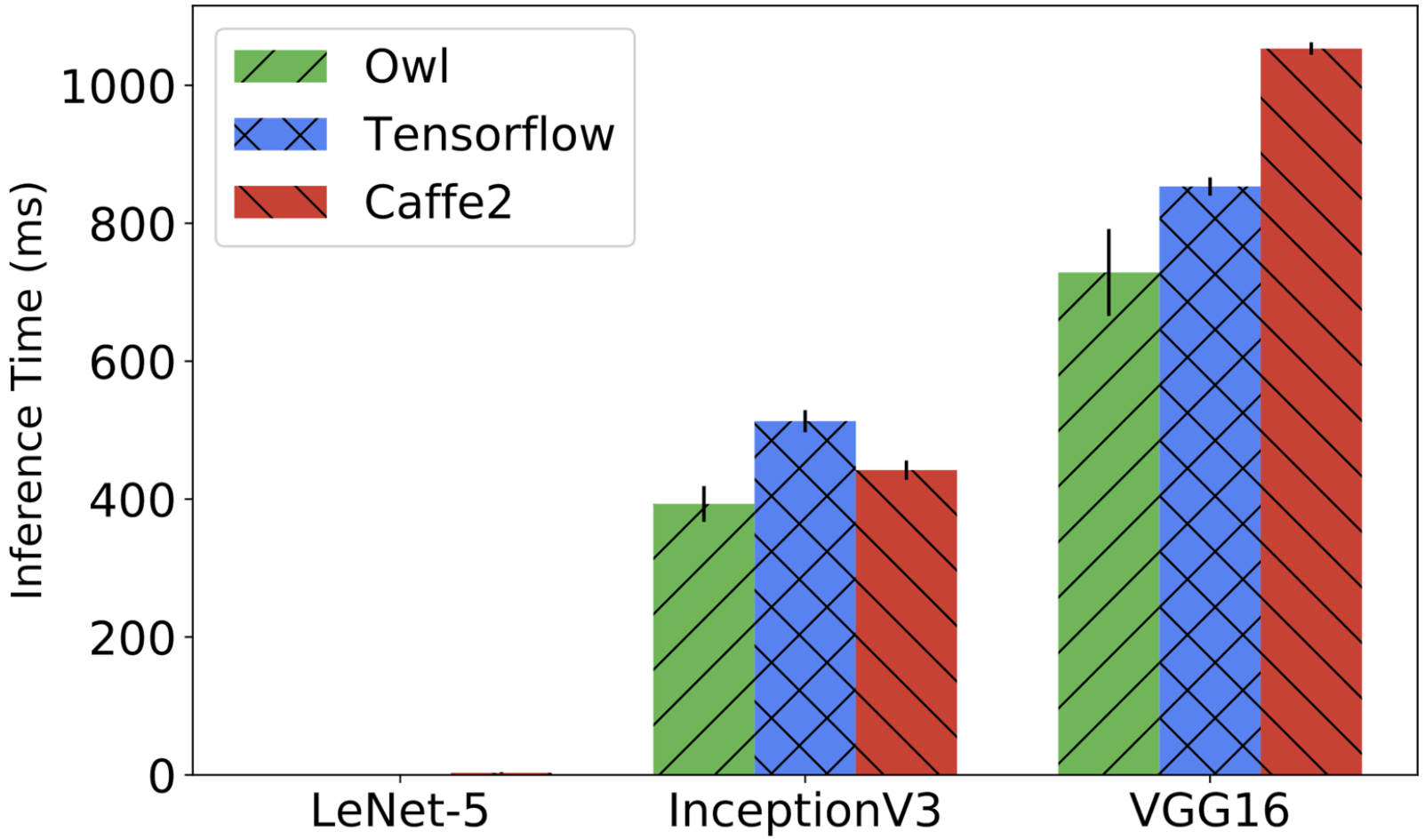



The random sampling is based on the fact that node identifiers are uniformly distributed in a name space.

Node can estimate the population size based on the allocated ID density in the name space.

UNIVERSITY OF CAMBRIDGE

# Owl's Performance

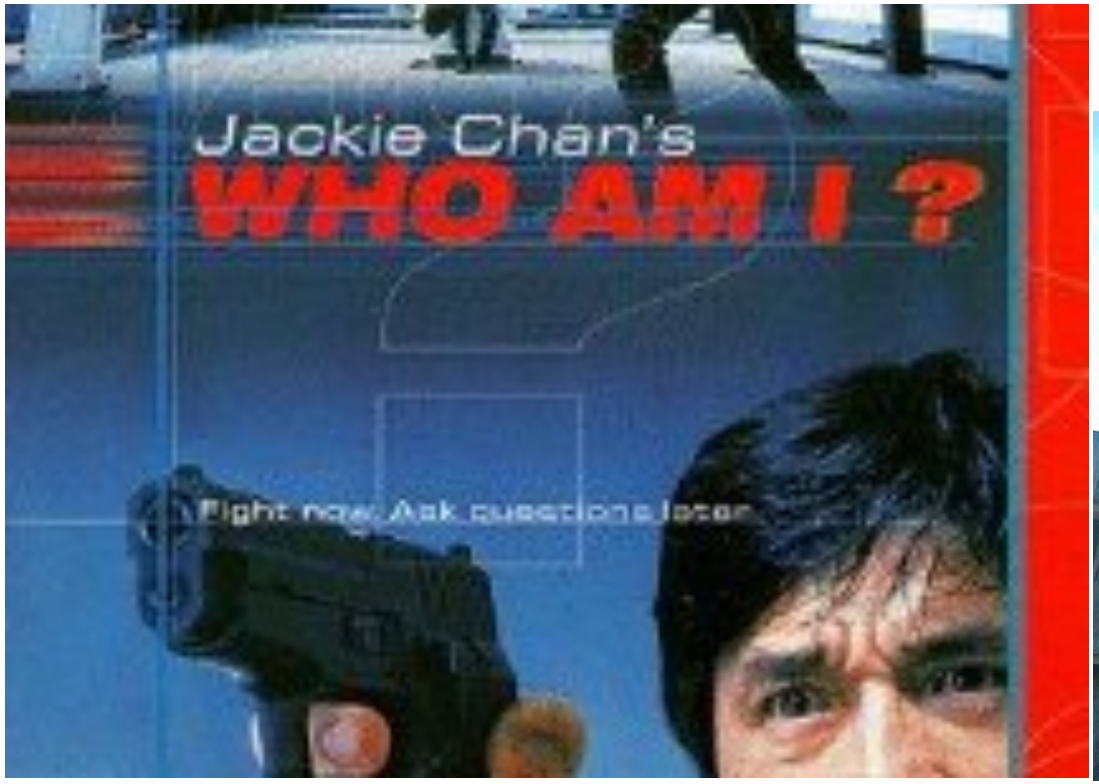



UNIVERSITY OF CAMBRIDGE

# Who Am I? & lets not speculate further ☺

**Thanks to EPSRC/databox**

- &Liang Wang, et al, Cambridge




**Thanks to Turing/Maru**

- &Peter Pietzuch, Imperial