

# Cloudy systems

— Taking the most out of the HPC Cloud



Workshop at UvA  
31<sup>st</sup> January 2018

Ander Astudillo <[ander.astudillo@surfsara.nl](mailto:ander.astudillo@surfsara.nl)>  
Niek Bosch <[niek.bosch@surfsara.nl](mailto:niek.bosch@surfsara.nl)>



# Recap: defining cloud computing

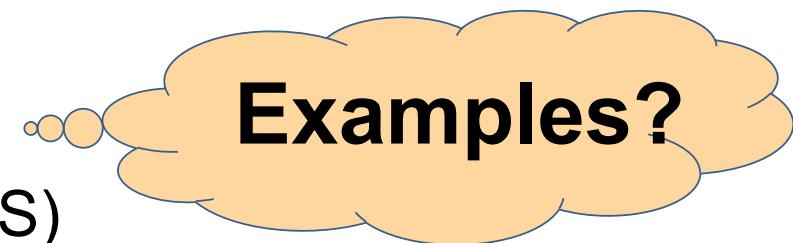
## Essential characteristics:

- On-demand **self-service**
- Broad **network access** (ubiquitous + convenient + on-demand)
- Resource pooling
- Rapid **elasticity** ⇒ 

- 1. Scaling
  - 2. API
- **Measured service**

## Service models:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)



# But why...?

## ...scaling

- Sequential run takes forever
- Not enough local resources (e.g.: memory)
- Analyse more data
- Achieve higher accuracy
- ...



**Examples?**

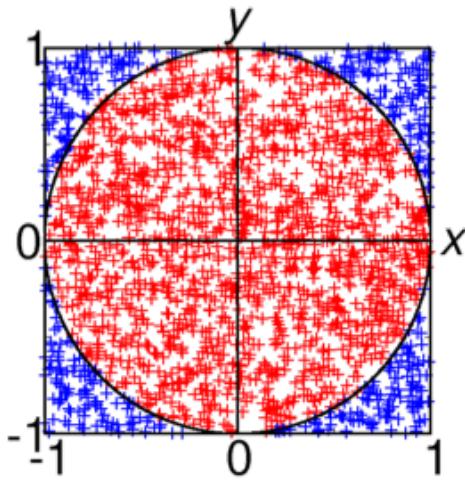
## ...elasticity

- Booking fixed resources in advance is:
  - A waste
  - Too expensive
  - Unpredictable
  - ...

# The naïve approach

Some programs are **already parallel**

- The end-user just needs to run them
- E.g.: Delft3D, XBeach, OpenFoam, Matlab...



Some problems are a matter of running the **same thing** (possibly) **with different parameters**

- You can simply run many of these runs independently at the same time on different computers
- E.g.: a Monte Carlo simulation

Embarassingly parallel problems!

# Agenda

1.- Scaling possibilities

2.- API overview

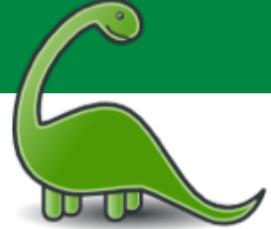
3.- Demo



# Scaling possibilities



# The concept (I)



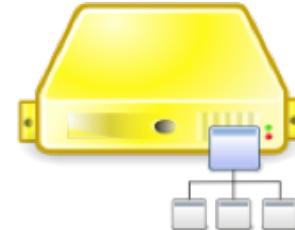
Your **application**  
may need more...



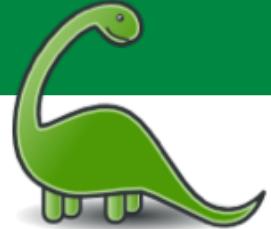
**Scale up**

vs.

**Scale out**



# The concept (II)



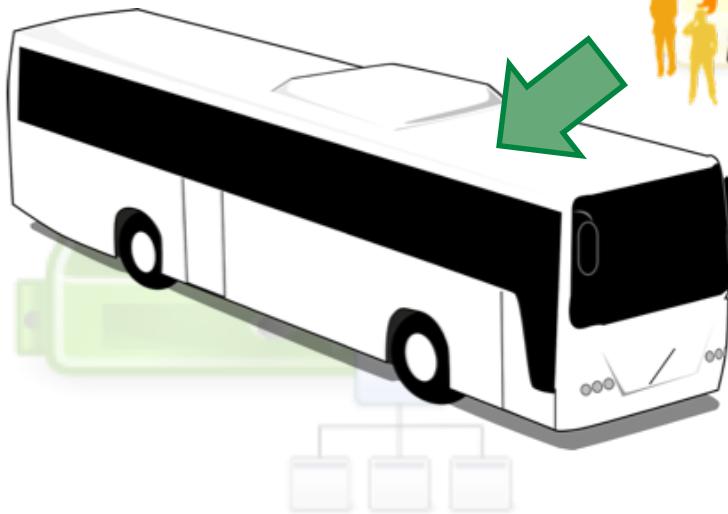
e.g. **transport people**  
may need more room...



**Scale up**

vs.

**Scale out**



# The concept (and III)



e.g. **transport people**

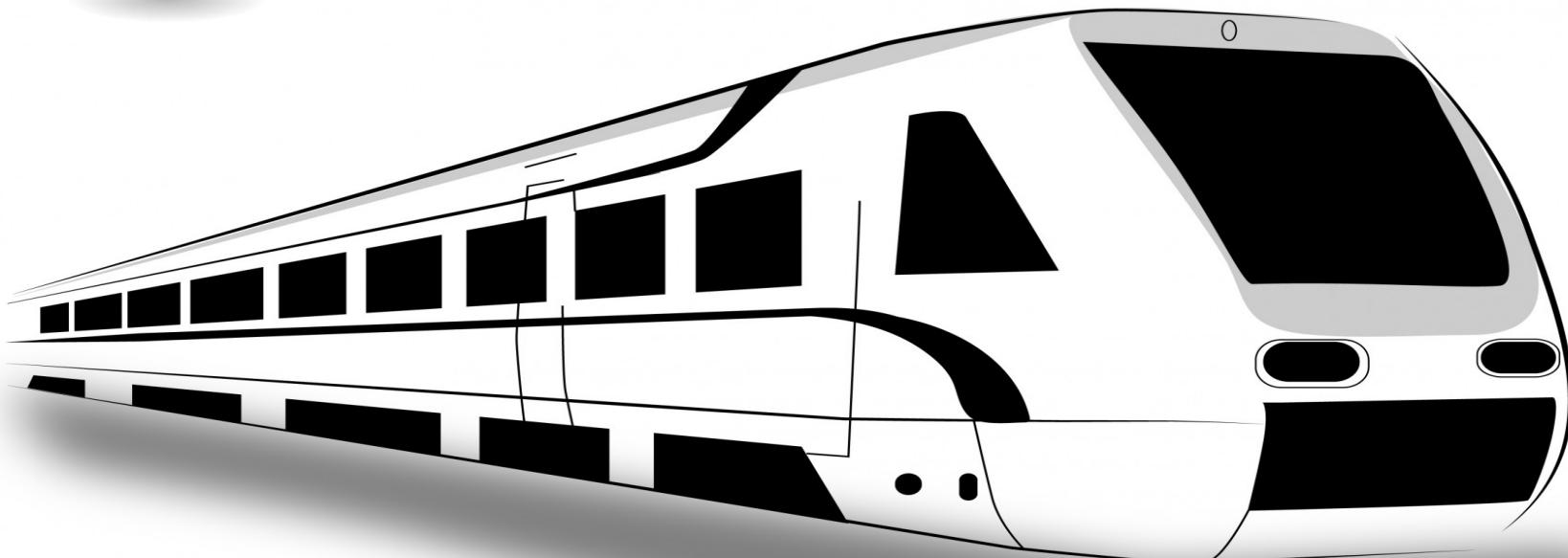
may need more room...



**Scale up**

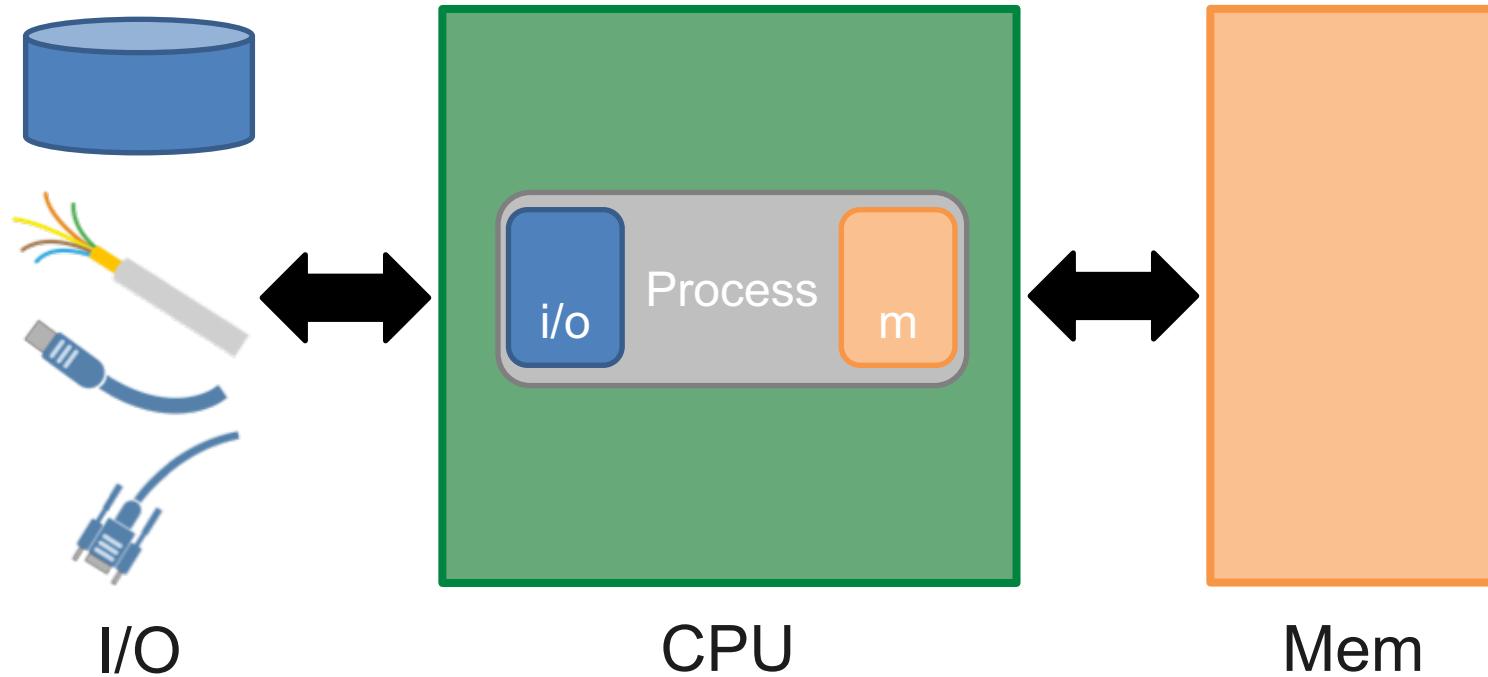
**AND**

**Scale out**



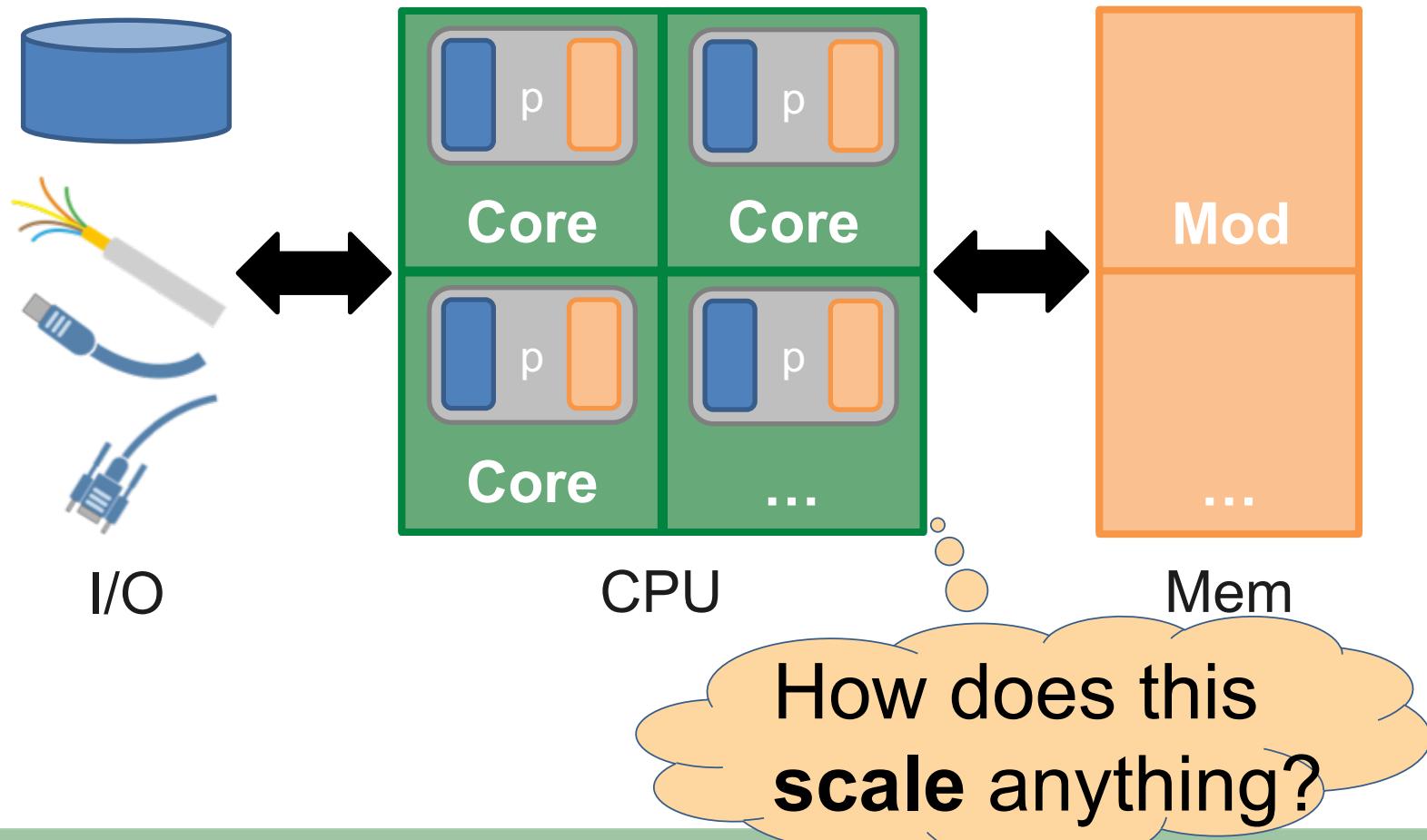
# Some theory (I)

## Meet: the CPU



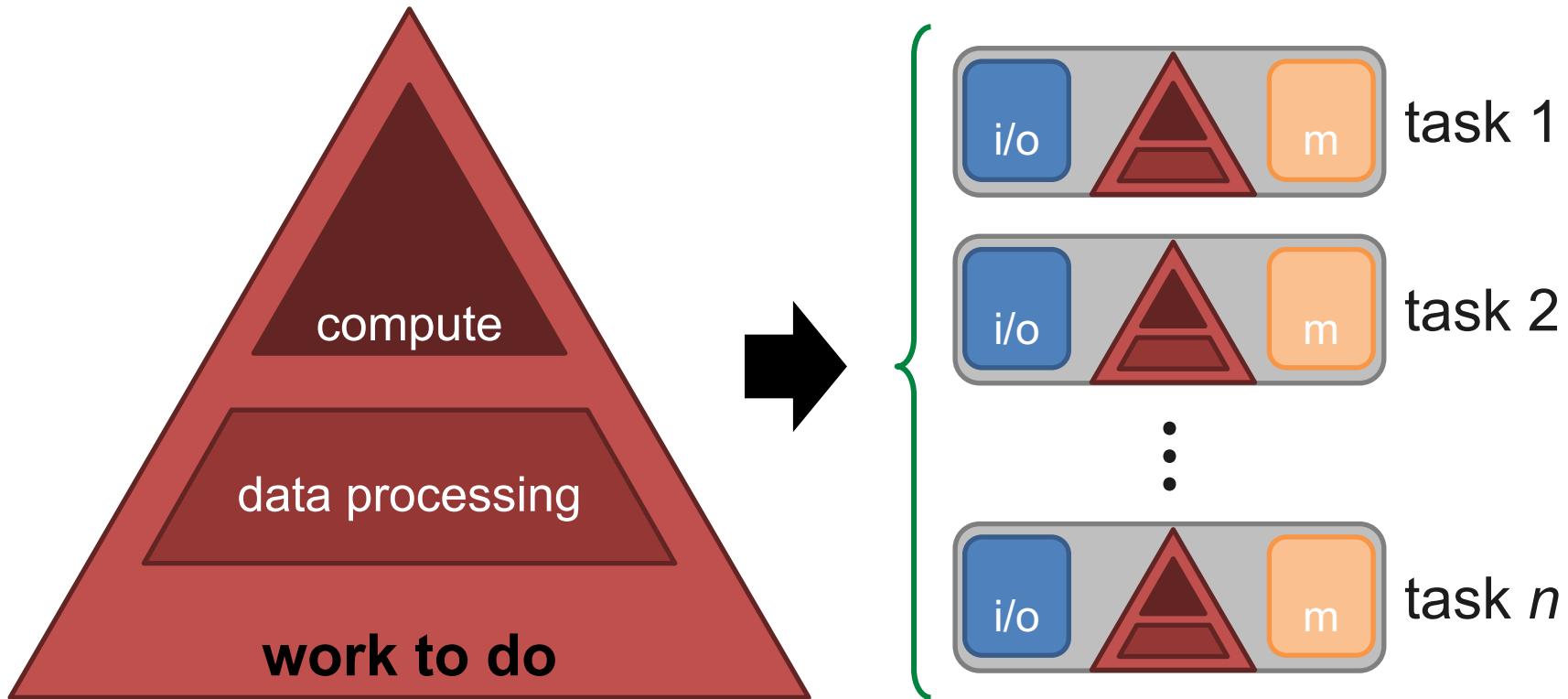
# Some theory (and II)

## Meet: parallel processing



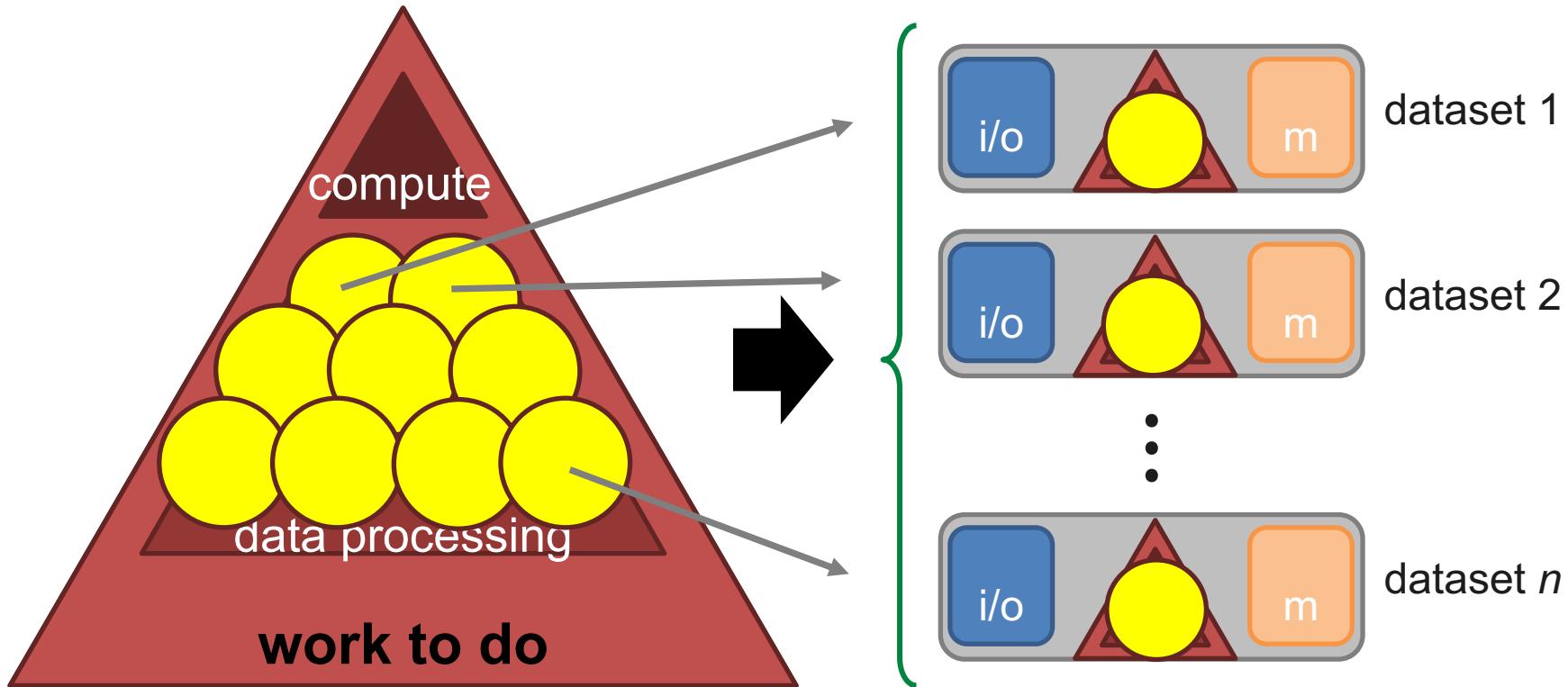
# Dividing work (I)

## Parallelism: task partitioning



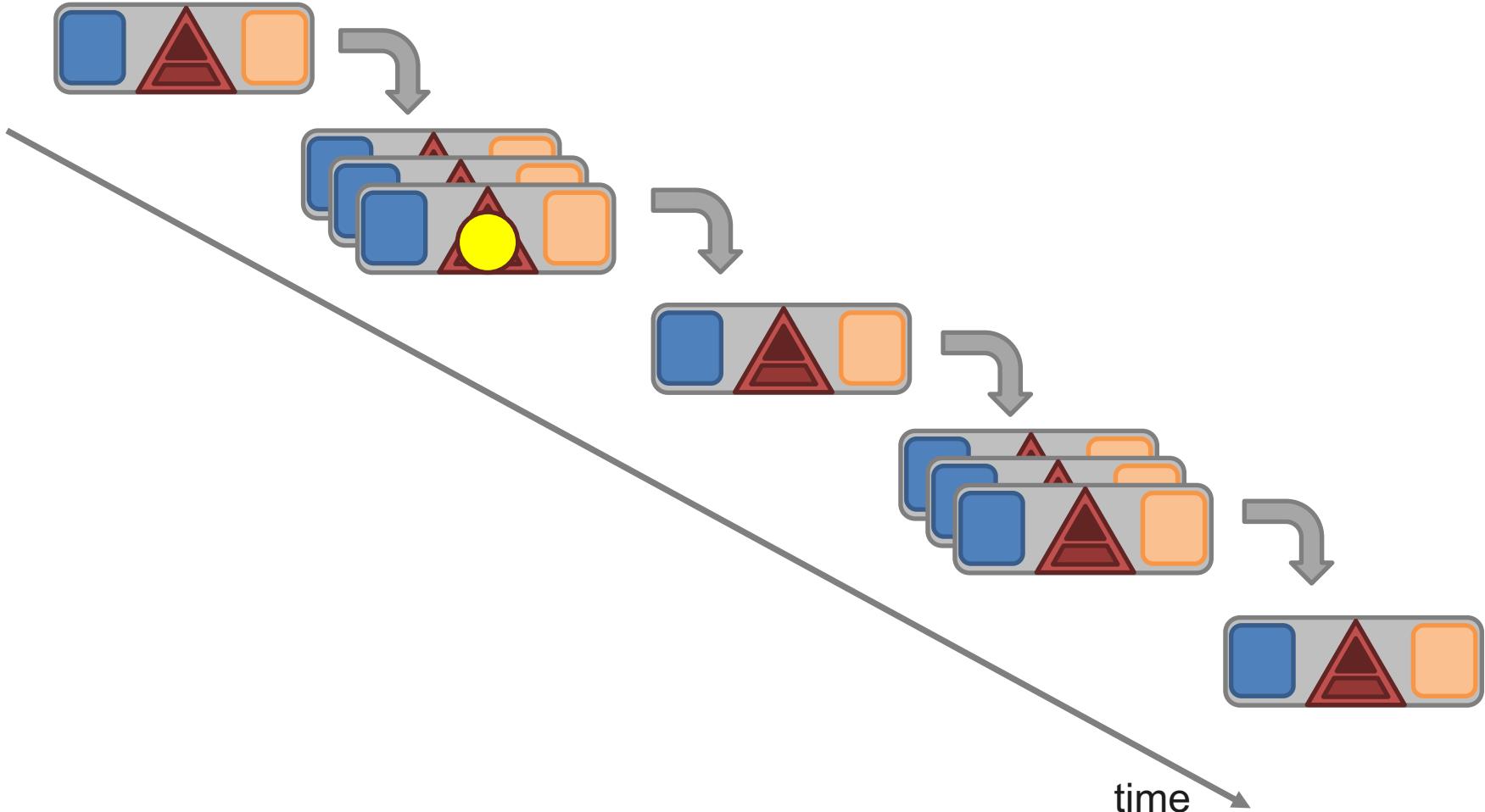
# Dividing work (II)

## Parallelism: **data** partitioning



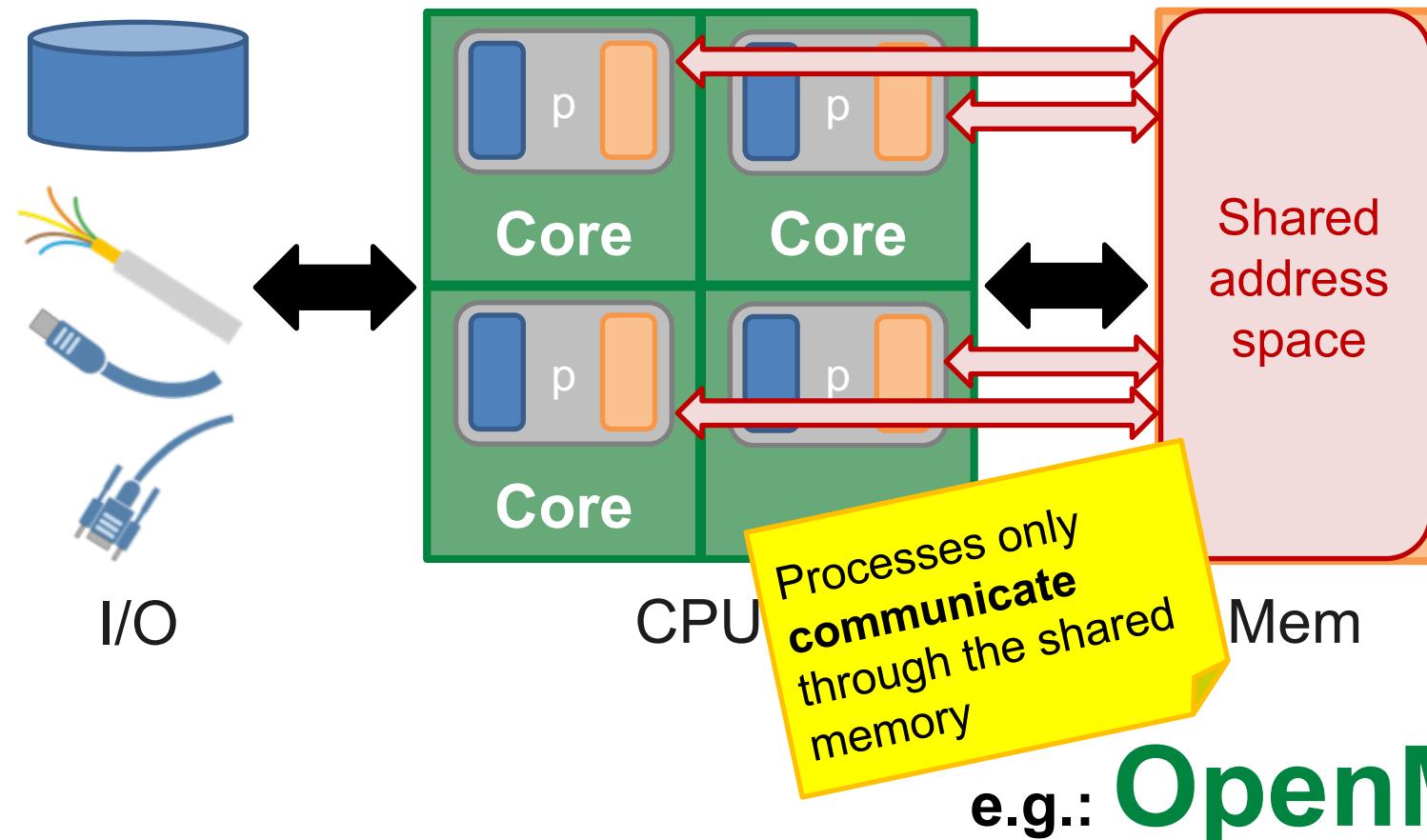
# Dividing work (and III)

Example: a possible parallel program (or workflow)



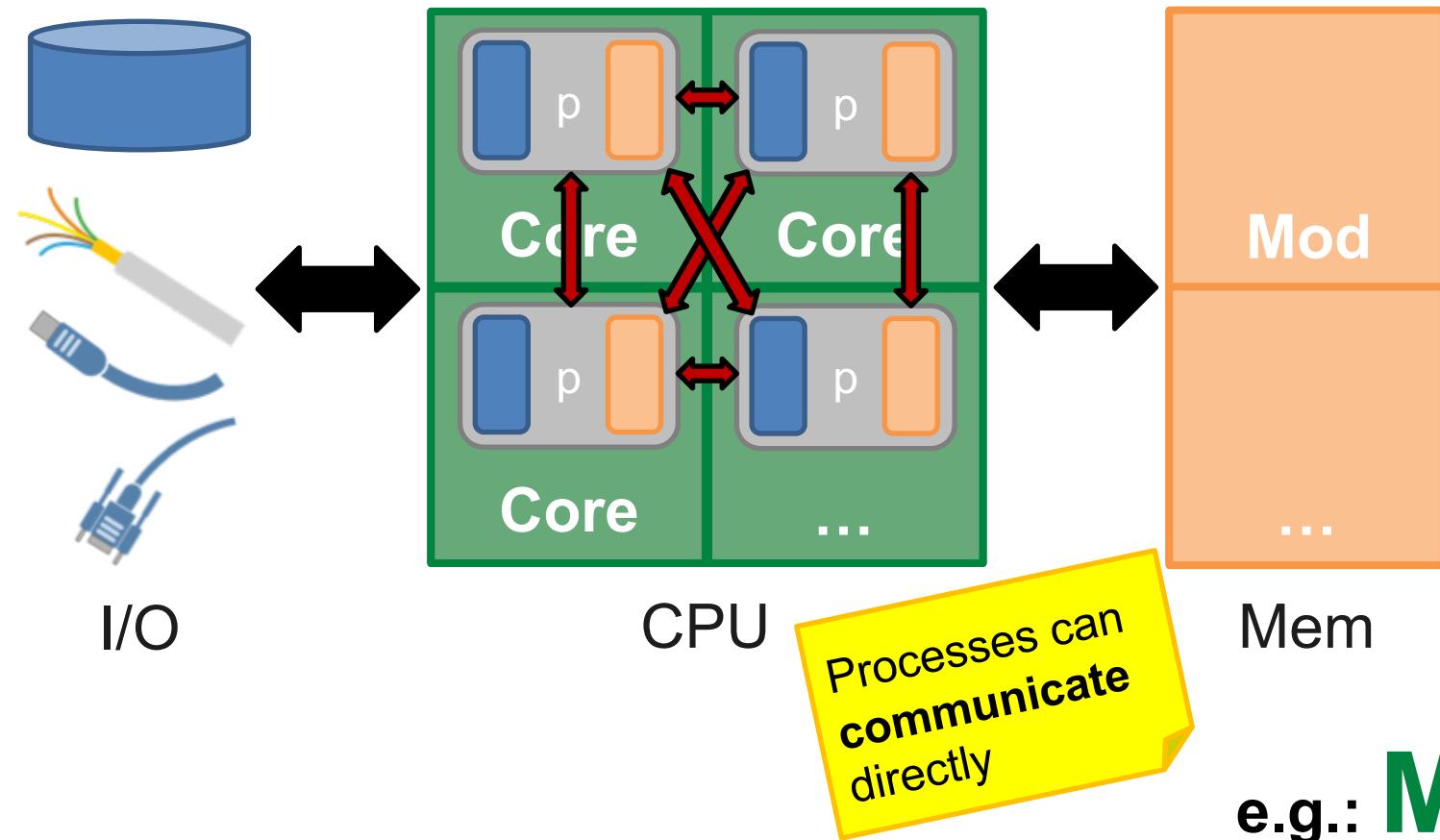
# Parallel programming (I)

## Technique: shared memory



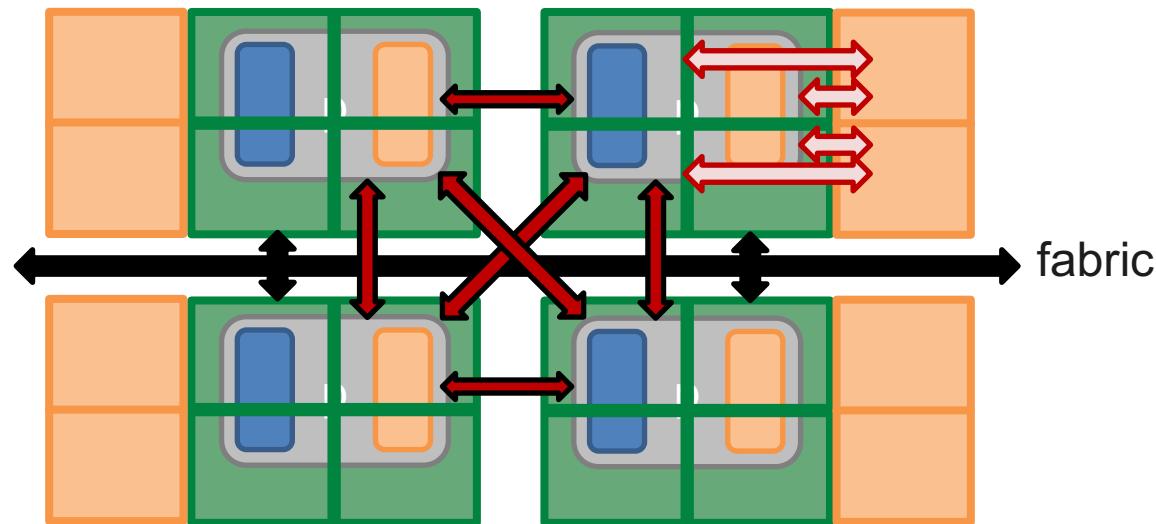
# Parallel programming (II)

## Technique: message-passing



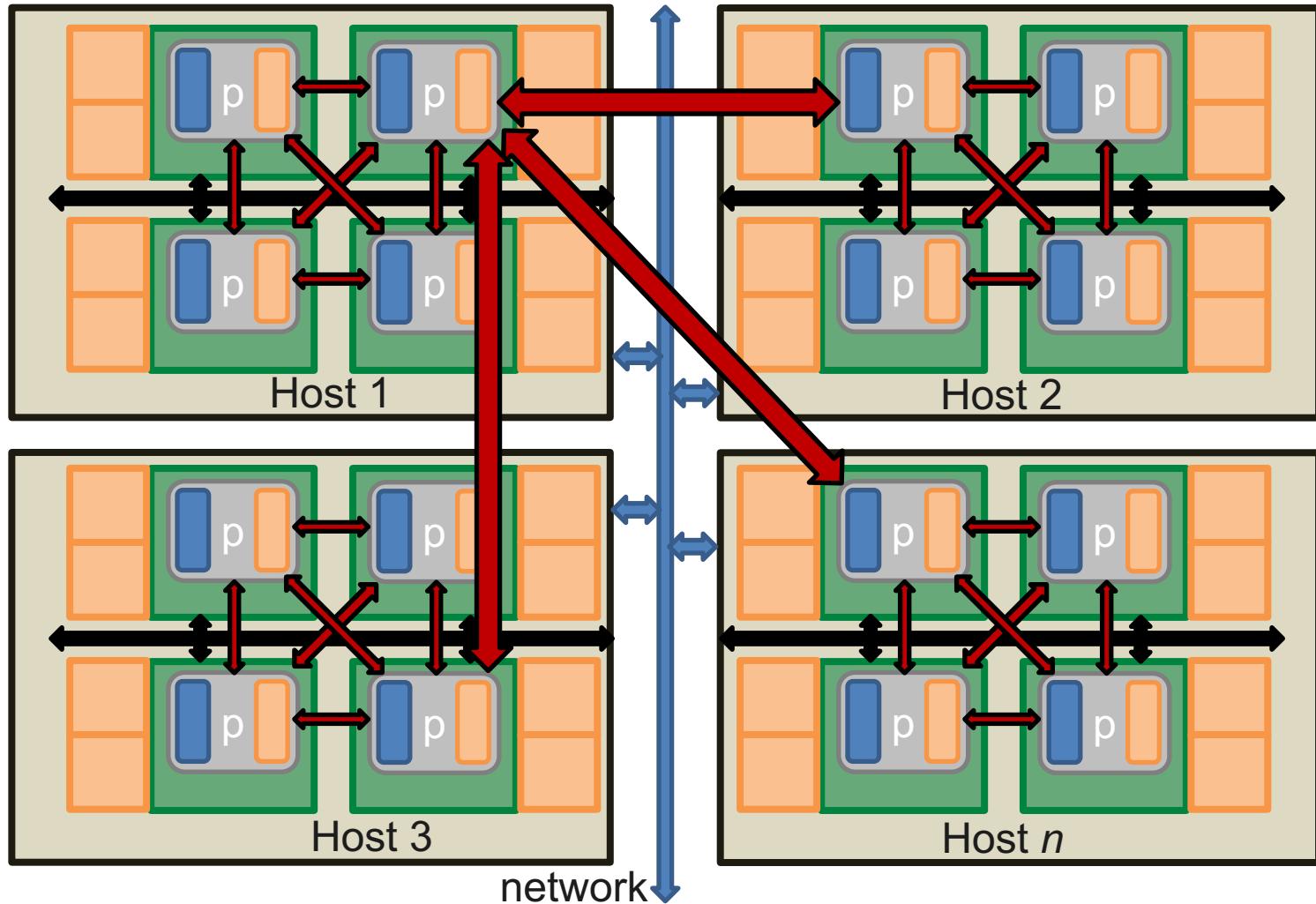
# Parallel programming (III)

## Combinations: shared memory and message passing



# Parallel programming (and IV)

## Combinations: multiple physical machines

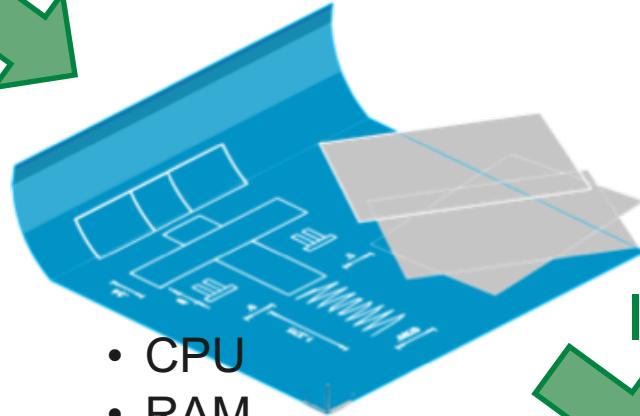


# IaaS: Your place to run VMs



- Data store
- Persistency
- ...

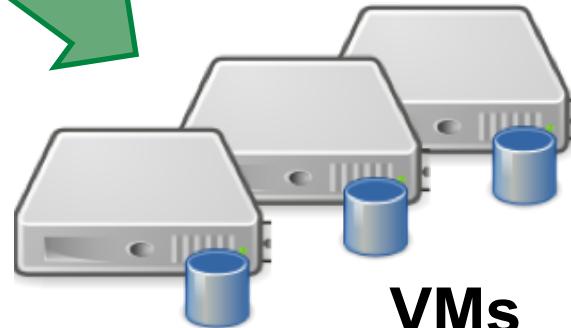
**Images**



- CPU
- RAM
- I/O
  - Disks
  - Network
- ...

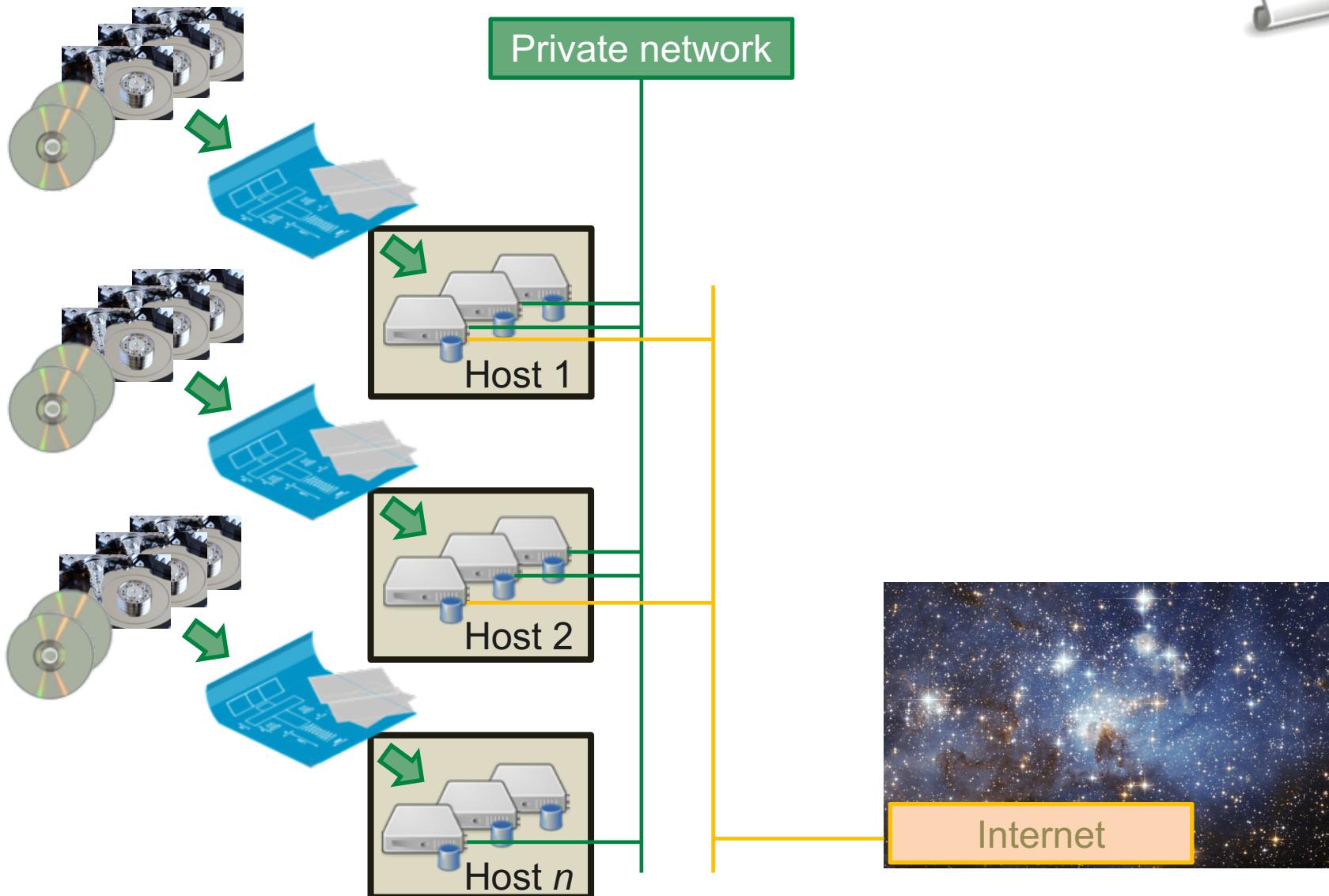
**Template**

**Instantiate**

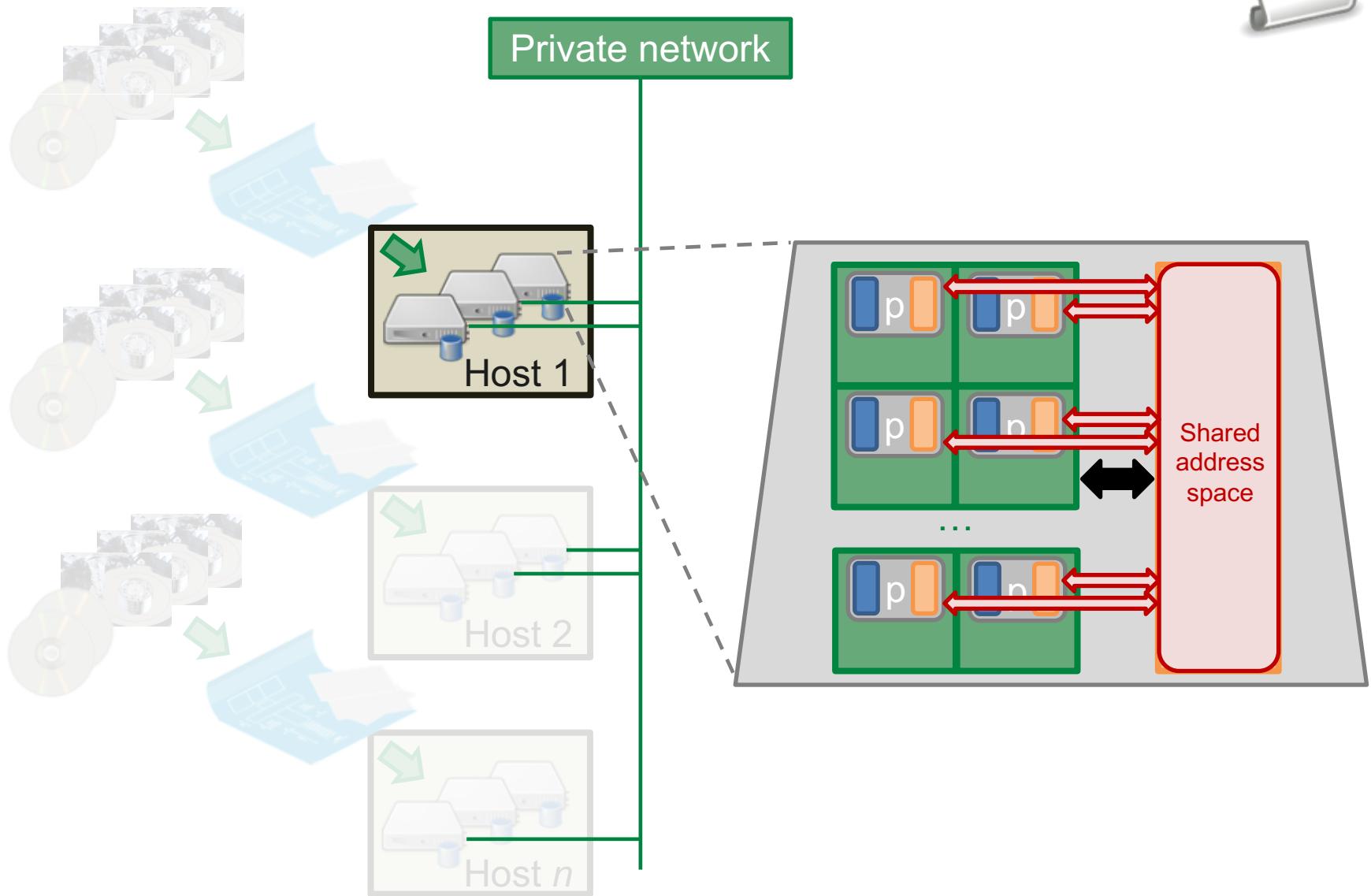


**VMs**

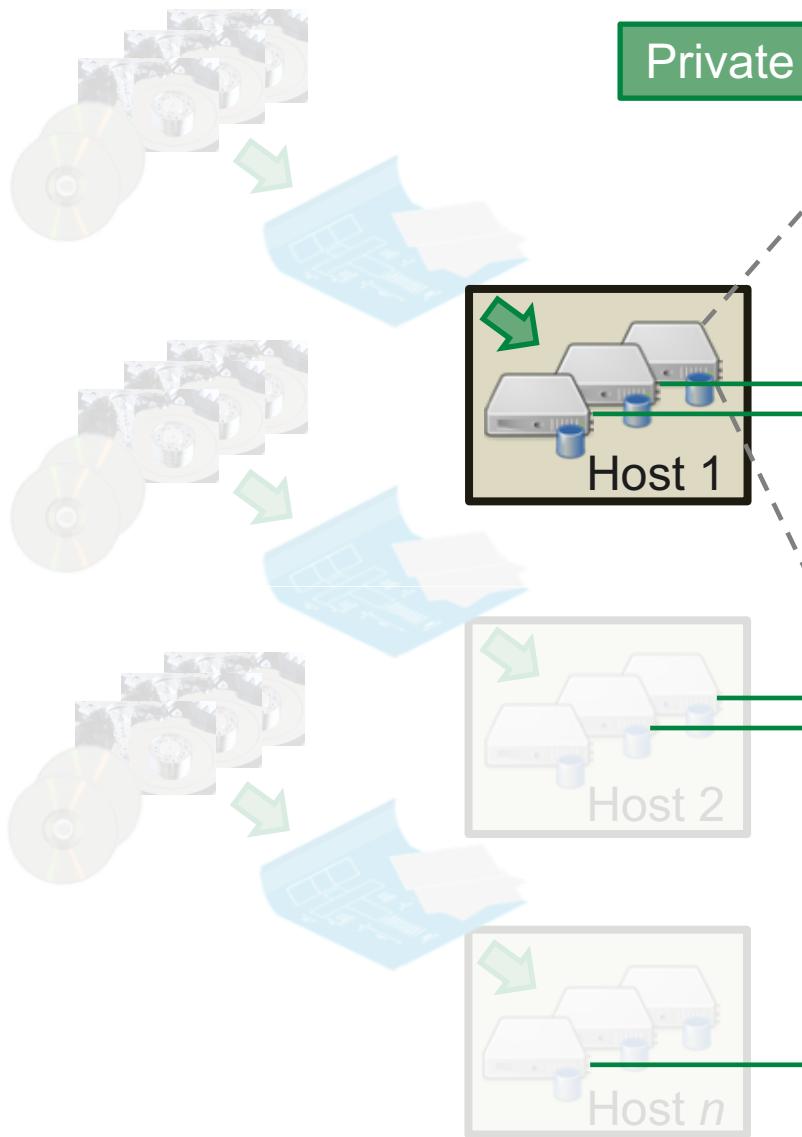
# IaaS: your interconnected VMs



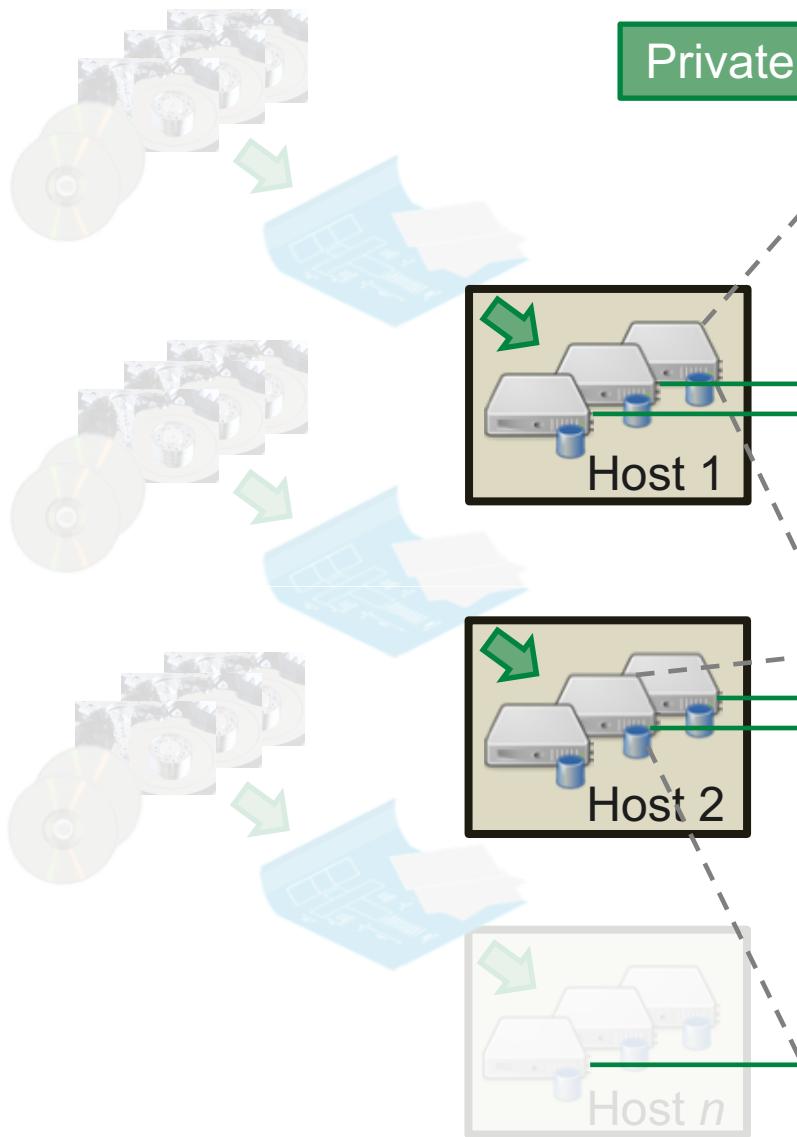
# Example: OpenMP



# Example: MPI



# Example: MPI



Private network

Host 1

Host 2

Host n

# Some thoughts

**Parallel programming can be tricky:**

- Need to know your **algorithm**
- Need to know your **data**
- Need to know your **architecture**

Try to **optimise**:

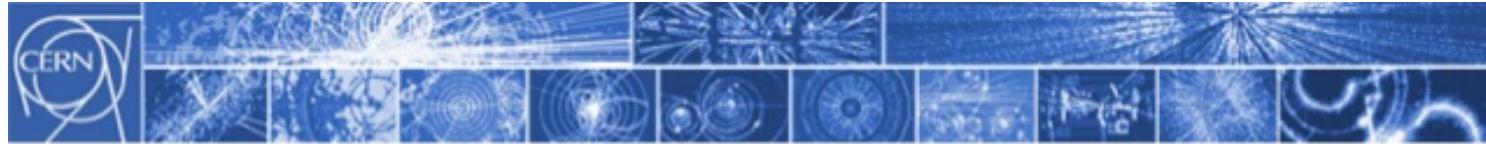
- Identify sequential **bottlenecks**
- Strive for **data** locality
- Identify **latencies**
- Minimise **communication**
- Be wary of **concurrency**:
  - **Deadlocks**
  - **Race conditions**
- Prepare for **failures**: machines, networks, timeouts...

So... you may as well be better off using a naïve approach! ☺

# API overview

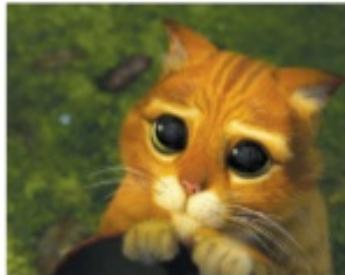


# Why automation?



## pets vs. cattle

Borrowed from  
@randybias at Cloudscaling  
<http://www.slideshare.net/randybias/the-cloud-revolution-cyber-press-forum-philippines>



- Pets are given names like `pussinboots.cern.ch`
- They are unique, lovingly hand raised and cared for
- When they get ill, you nurse them back to health



- Cattle are given numbers like `vm0042.cern.ch`
- They are almost identical to other cattle
- When they get ill, you get another one

- Future application architectures should use Cattle but Pets with strong configuration management are viable and still needed

Gavin McCance, CERN

17



OpenNebula

## XML-RPC over http

- bindings for Java, Ruby (also Python, NodeJS...)
- **Methods** like `one.<object>.<action>`
  - e.g.: `one.vm.rename`
  - **Pools**, like: `one.vmpool.info`
- **Parameters**, position-based
- **Output**, a 3-tuple (A, B, C) where:
  - A: correct or error response
  - B: returned info (if correct);  
error message (if error)
  - C: numeric error code

Operate/query on:

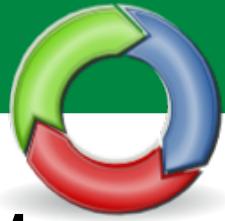
- Images
- Templates
- Virtual Machines
- Quotas
- ...

# Demo

3



# Example (I)



## List my VMs

```
class VmList:  
    """A simple list of my VMs"""\n    ONE_ENDPOINT = 'http://ui.hpccloud.surfsara.nl:2633/RPC2'\n    ONE_USER = 'username' # replace this with yours\n    ONE_PASS = 'pass'     # replace this with yours\n    def __init__(self):\n        self.client = oca.Client(\n            self.ONE_USER + ':' + self.ONE_PASS, self.ONE_ENDPOINT)\n\n    def fetch_vms(self):\n        xml_string = self.client.call('vmpool.info', -3, -1, -1, -2)\n        root = ET.fromstring(xml_string)\n        return root\n\nif __name__ == '__main__':\n    xml = VmList()\n        .fetch_vms()\n    print(XmlUtil\n        .prettyify(xml))
```

one.vmpool.info		
• Description: Retrieves information for all or part of the VMs in the pool.		
• Parameters		
Type	Data Type	Description
IN	String	The session string.
IN	Int	Filter flag - <= -3: Connected user's resources - -2: All resources - -1: Connected user's and his group's resources - > = 0: UID User's Resources
IN	Int	When the next parameter is >= -1 this is the Range start ID. Can be -1. For smaller values this is the offset used for pagination.
IN	Int	For values >= -1 this is the Range end ID. Can be -1 to get until the last ID. For values < -1 this is the page size used for pagination.
IN	Int	VM state to filter by.

# Example (and II)



List my VMs  
(output)

```
<VM_POOL>
  <VM>
    <ID>164</ID>
    <UID>247</UID>
    <GID>108</GID>
    <UNAME>ander</UNAME>
    <GNAME>workshop</GNAME>
    <NAME>Ubuntu-15.04</NAME>
    ...
    <LCM_STATE>3</LCM_STATE>
    <TEMPLATE>
      <CPU>...</CPU>
      ...
      </TEMPLATE>
    </VM>
    <VM>...</VM>
    ...
  </VM_POOL>
```

**Request:** <https://e-infra.surfsara.nl>  
**UI:** <https://ui.hpccloud.surfsara.nl>  
**Doc:** <https://doc.hpccloud.surfsara.nl>

## Credits

Images: Wikipedia, Science Park, RRZE icons, NIST, nVidia, Ceph, publicdomainpictures.net, publicdomainvectors.org, cs.unc.edu/~weicheng  
Slides: SURFsara colleagues, CERN



Ander Astudillo  
<ander.astudillo@surfsara.nl>

Niek Bosch  
<niek.bosch@surfsara.nl>



<<EOF

# A ppendix



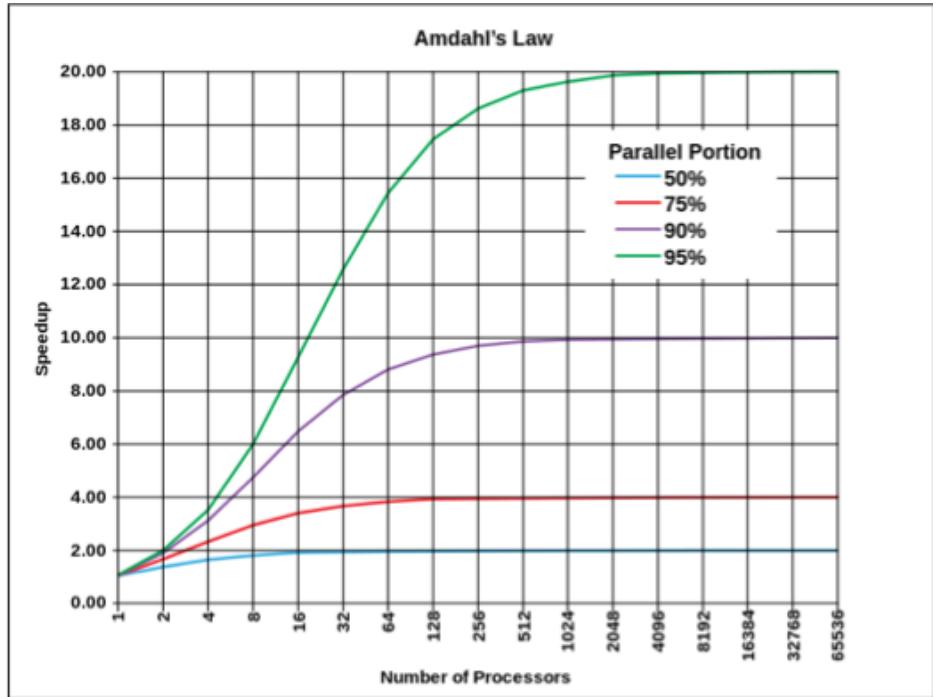
# Amdahl's law

$$T(s) = (1 - p)T + \frac{p}{s}T.$$

- $T(s)$ : running time after an improvement of  $s$
- $s$ : speedup factor of parallel part
- $p$ : % of the program that is parallel
- $T$ : original running time
- $W$ : fixed workload

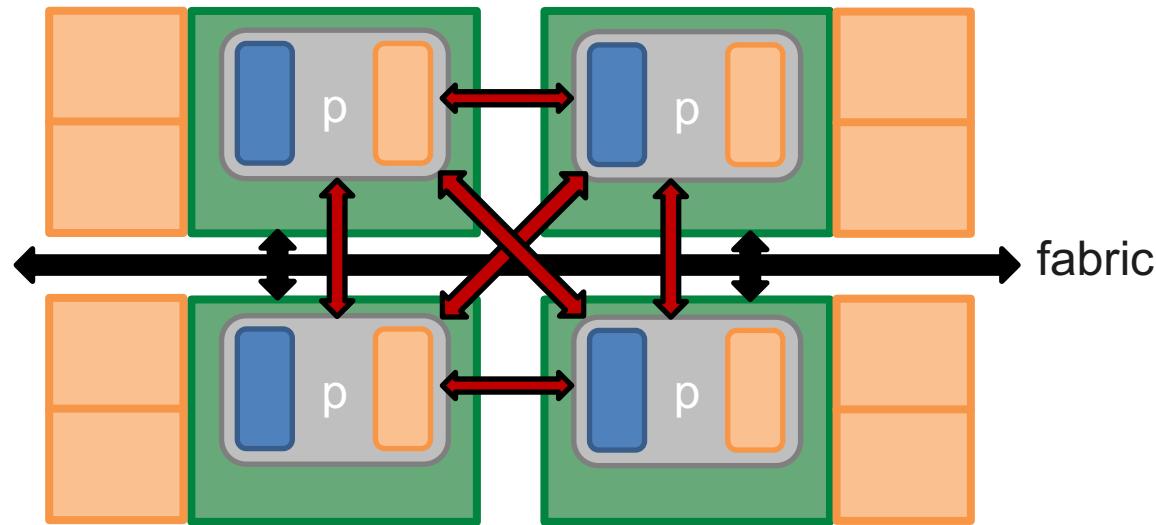
It's mainly the **algorithm** that defines speedup; rather than the amount of processors

$$S_{\text{latency}}(s) = \frac{TW}{T(s)W} = \frac{T}{T(s)} = \frac{1}{1 - p + \frac{p}{s}}.$$



Speedup is limited by the serial part of the program. E.g., if 95% of the program can be parallelised, the theoretical maximum speedup using parallel computing would be 20 times.

# NUMA



Processes  
communicate  
over the fabric