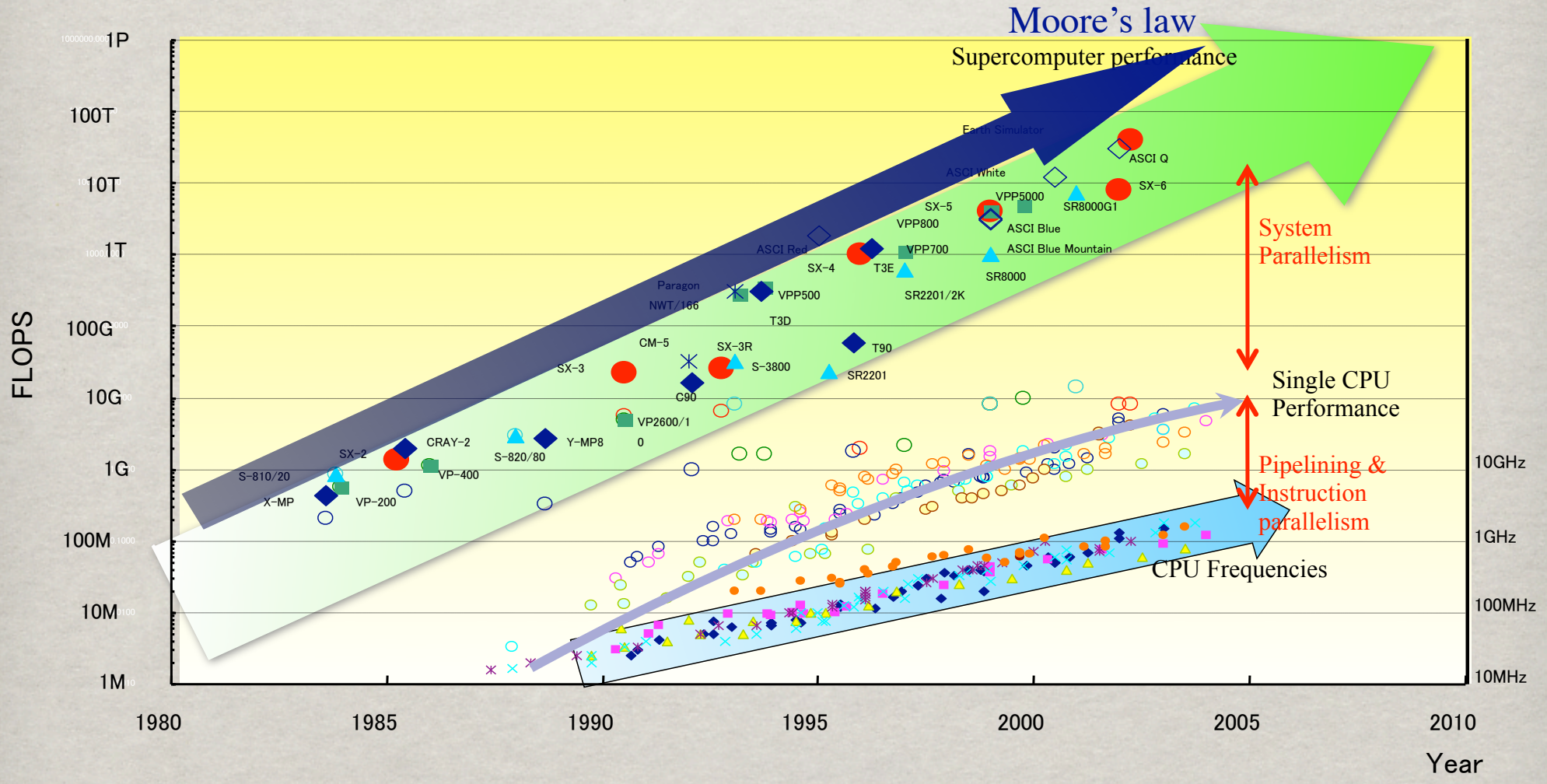
Parallel programming

Day 1
Distributed computing

Logistics

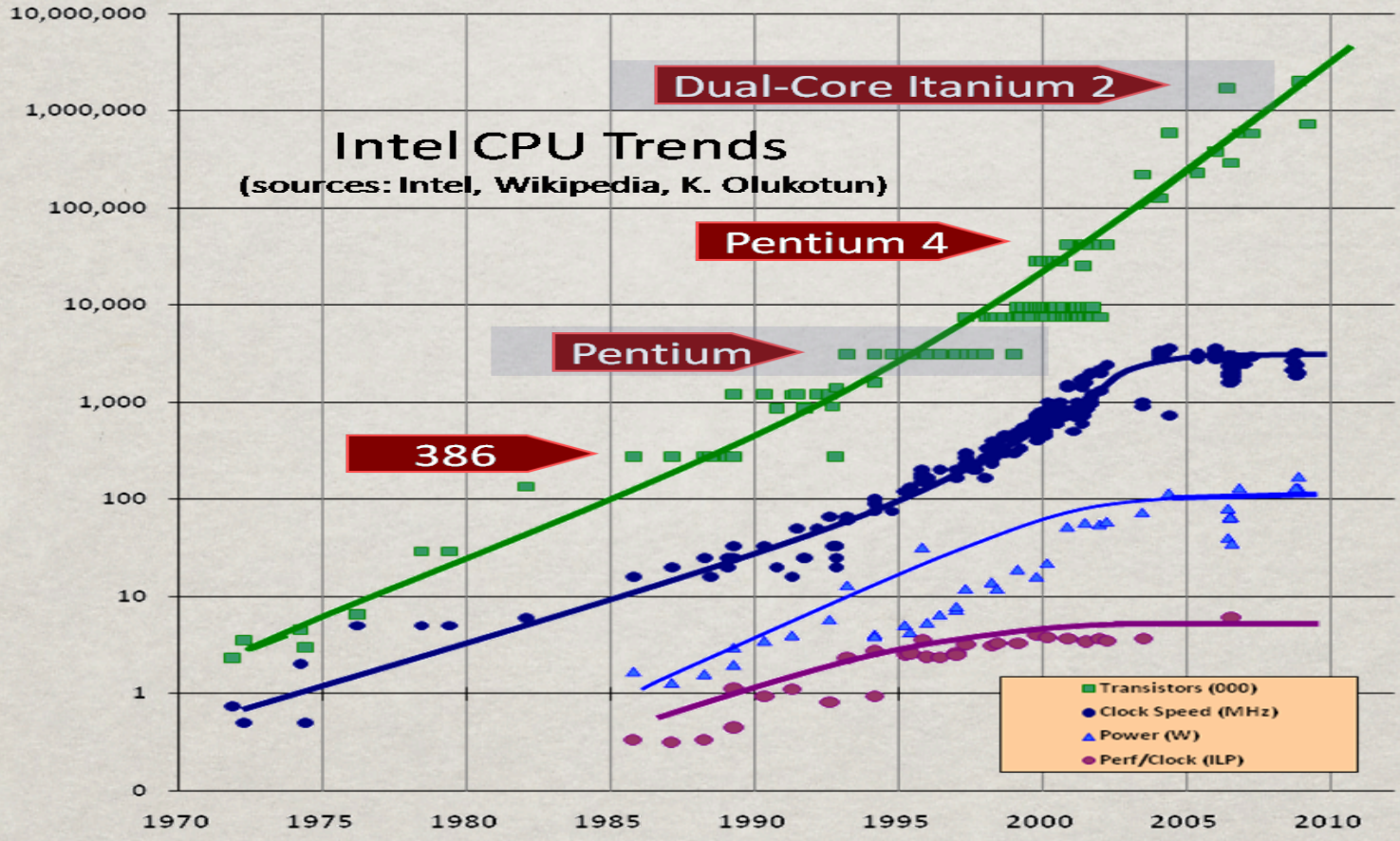
- 10:00 - 12:00 : lectures & questions
- 13:00 - 17:00 :
  practicals & questions & discussion
- Assistants: Roeland Douma & Roy Bakker

Trends



Data courtesy Jack Dongarra

Trends


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)

Why parallel programming?

- Computing needs: web apps, weather, medical research, gaming, astrophysics, etc.
- Single processor computers do not get faster fast enough to match the needs
- Goals:
  - Maximize **throughput** : computations / second
  - Minimize overall cost :
    maximize computations / euro
  - NB: energy is a form of cost

Performance vs throughput

Performance vs throughput

- **performance** = seconds / computation

Performance vs throughput

- **performance** = seconds / computation
- **throughput** = computations / second

Performance vs throughput

- **performance** = seconds / computation
- **throughput** = computations / second
- What's the difference?

Performance vs throughput

- **performance** = seconds / computation
- **throughput** = computations / second
- What's the difference?
  - Performance very constrained by hardware

Performance vs throughput

- **performance** = seconds / computation
- **throughput** = computations / second
- What's the difference?
  - Performance very constrained by hardware
  - Throughput is more flexible, but requires that **problems can be decomposed** into independent computations

Performance vs throughput

- **performance** = seconds / computation
- **throughput** = computations / second
- What's the difference?
  - Performance very constrained by hardware
  - Throughput is more flexible, but requires that **problems can be decomposed** into independent computations
  - Limits: Amdahl's and Gustafson's laws (check them out!)

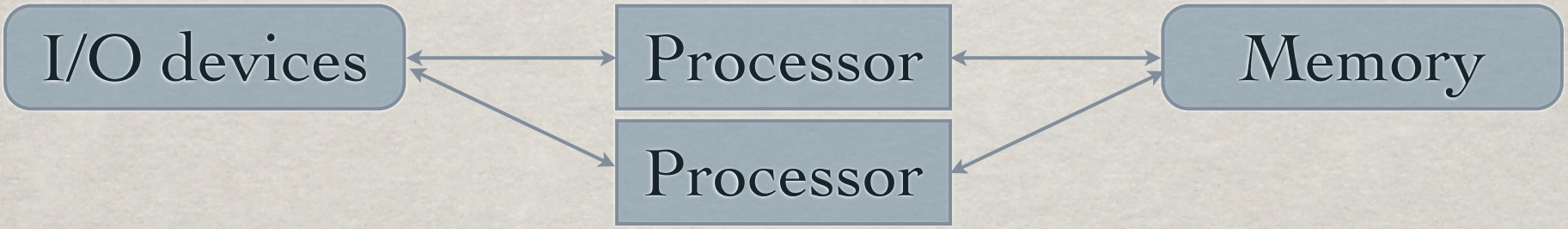
Part 1: platforms

Computers

This is a computer:

This too:

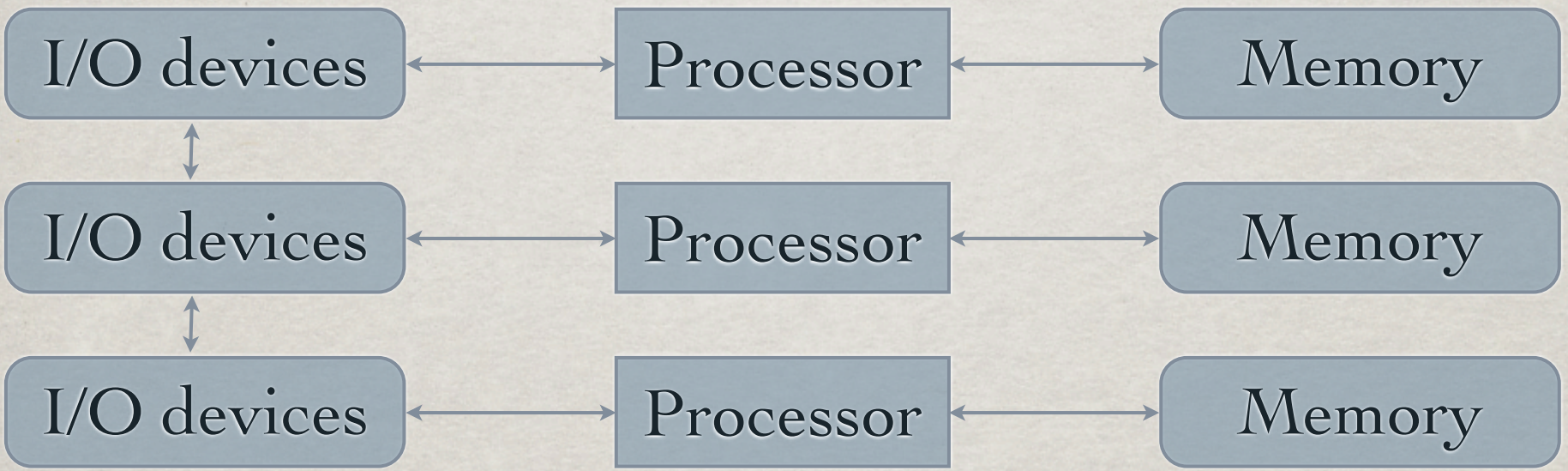
This too:



Can you recognize examples of each?

Computers

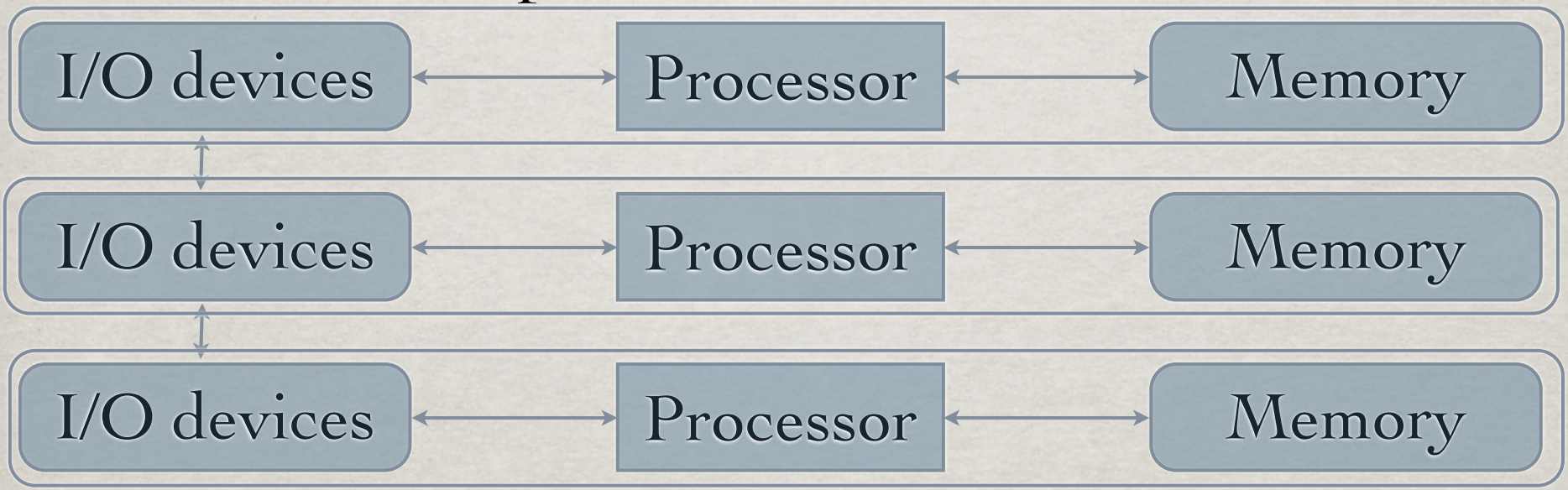
This is also a "computer":



... but is commonly called a **distributed system**

Computers

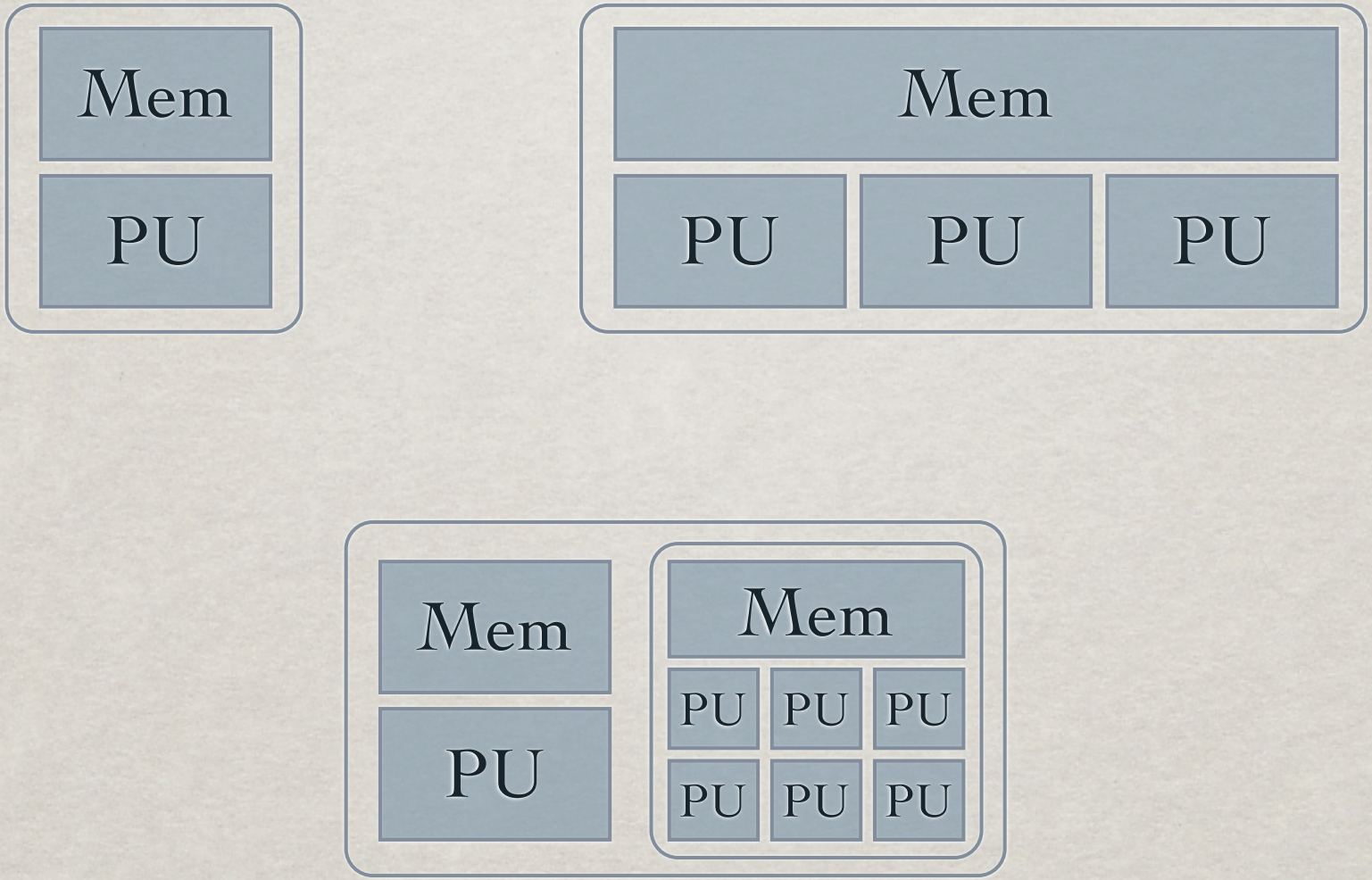
This is also a "computer":



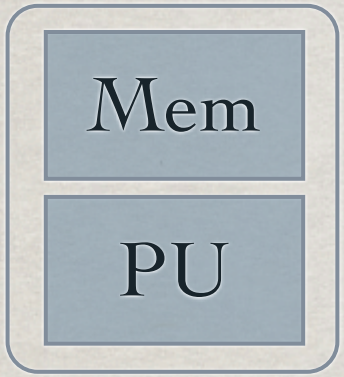
... but is commonly called a **distributed system**

"Node" =
group of PU(s) + Memory(ies) sharing an I/O interface

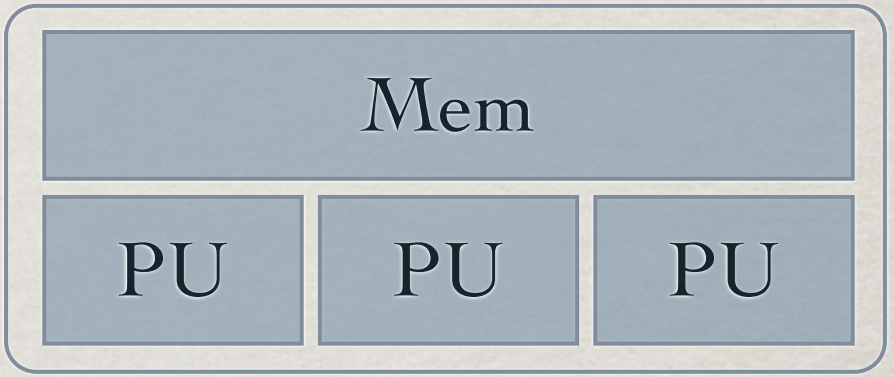
Topologies

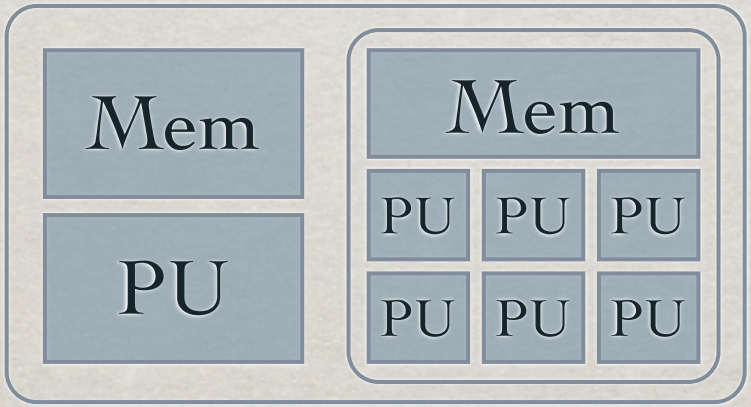


Topologies

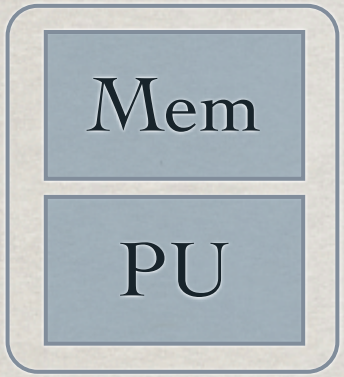


Uni-processor

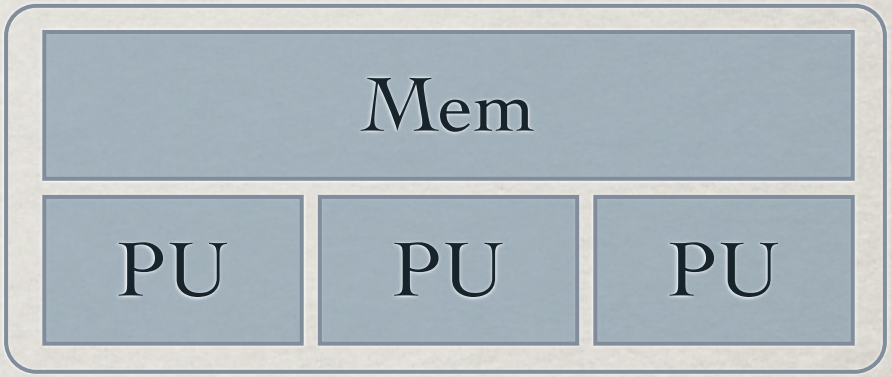




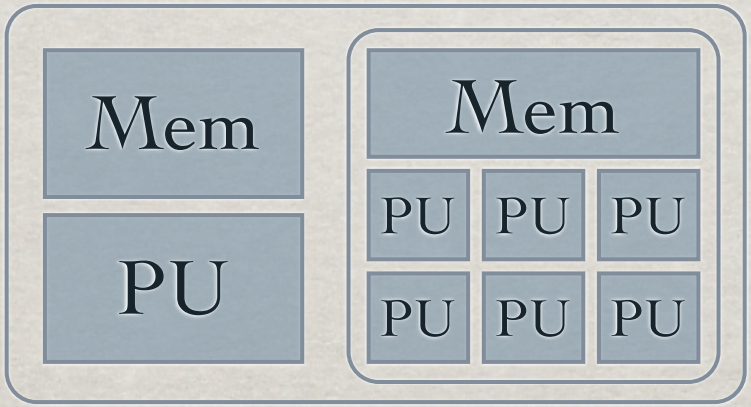
Topologies



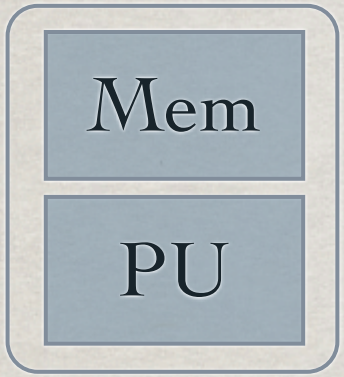
Uni-processor



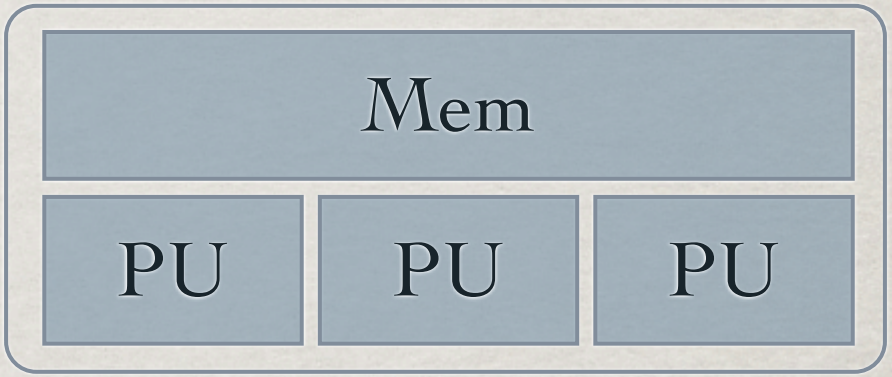
Multi-processor
Multi-core



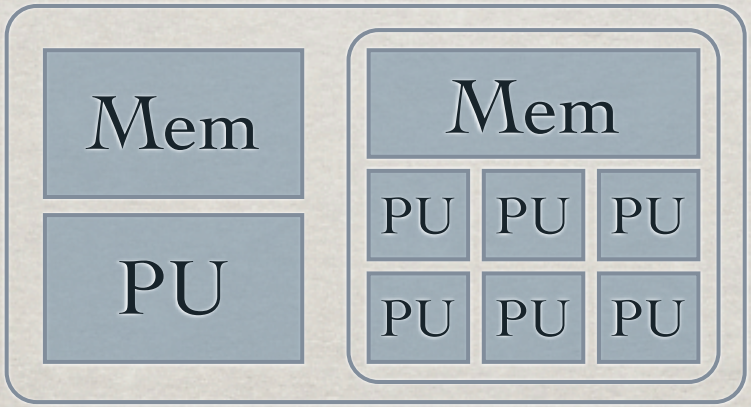
Topologies



Uni-processor



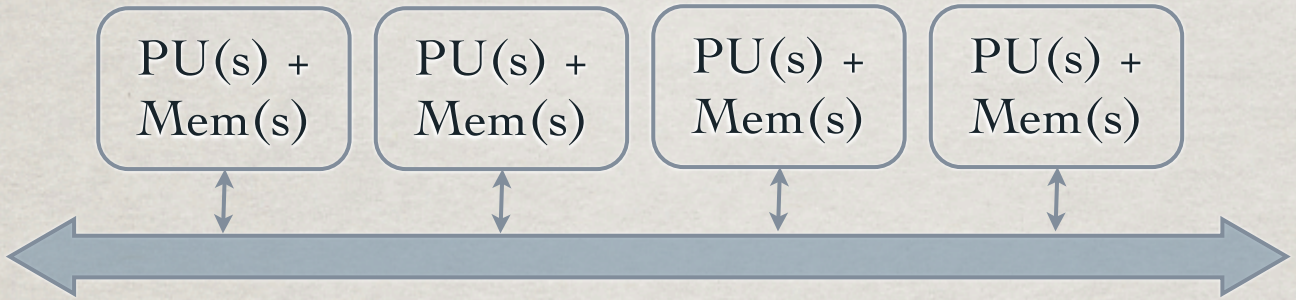
Multi-processor
Multi-core

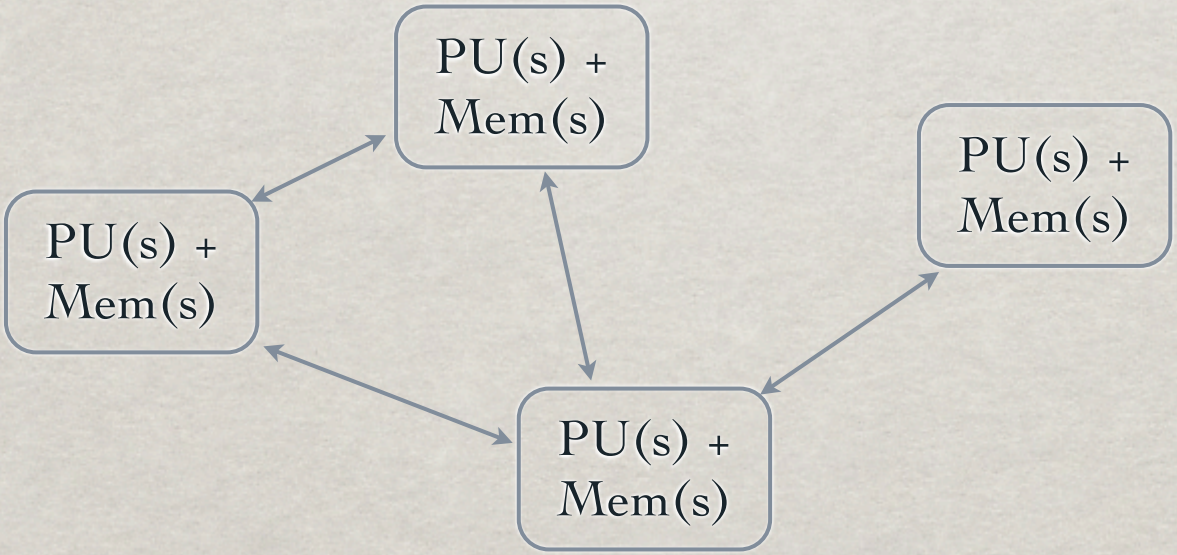


Heterogeneous multi-cores
Accelerators

Topologies for distributed systems

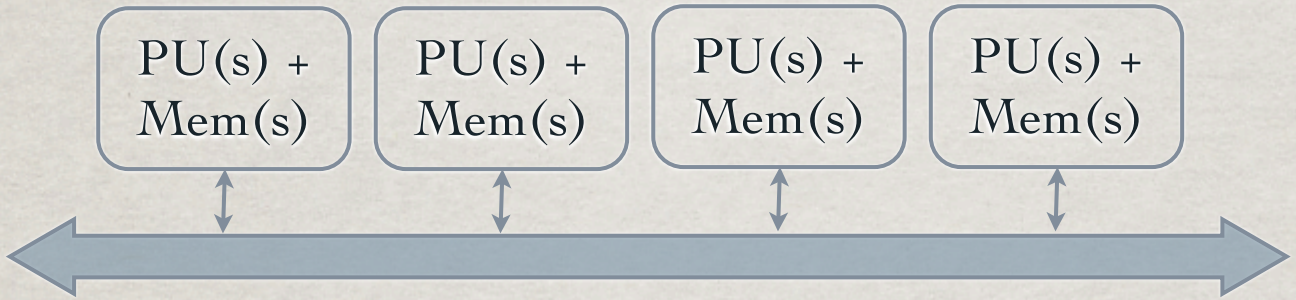
"nodes"



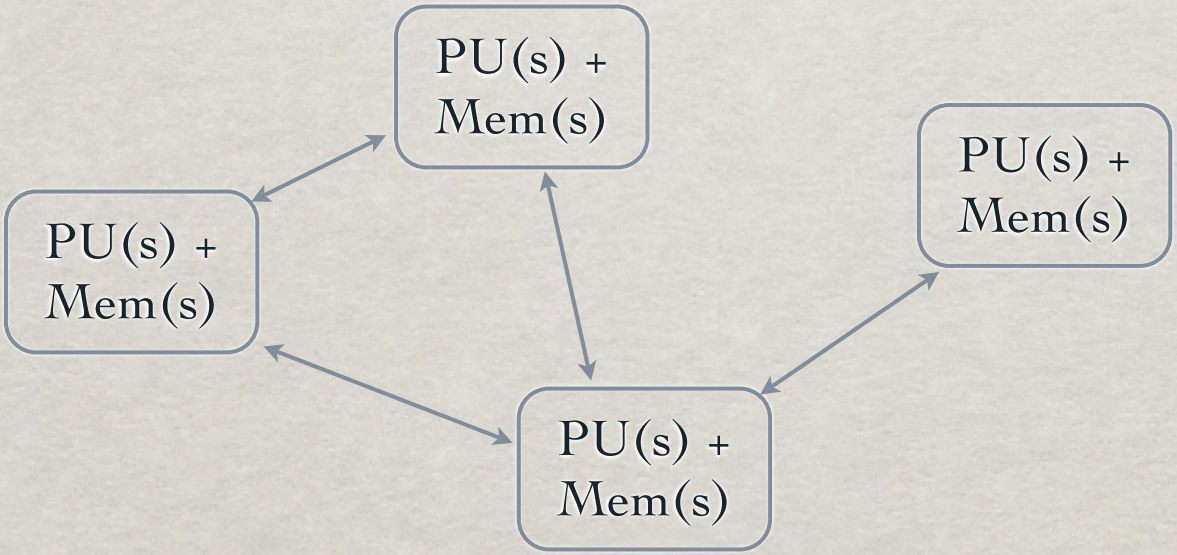


Topologies for distributed systems

"nodes"

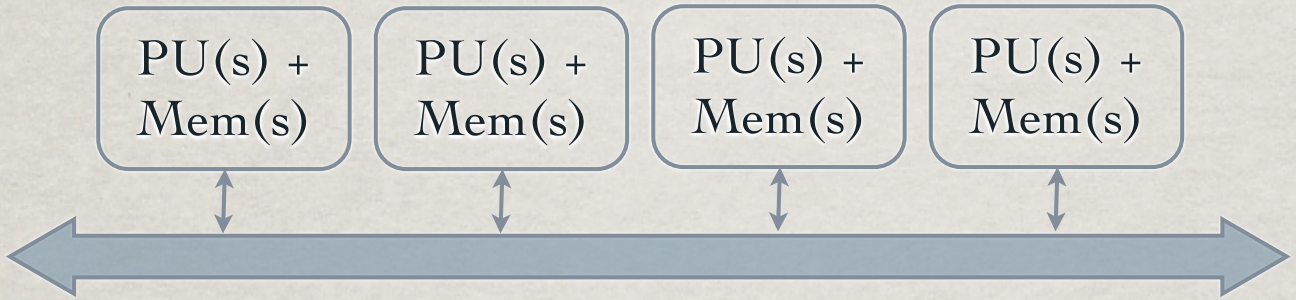


Regular interconnect: clusters, grids

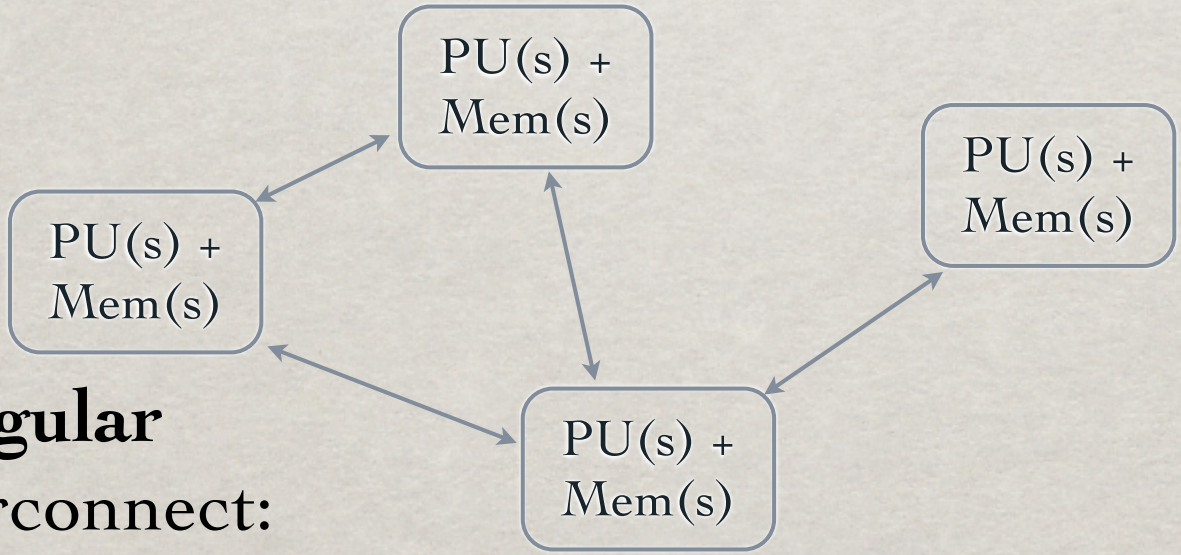


Topologies for distributed systems

"nodes"



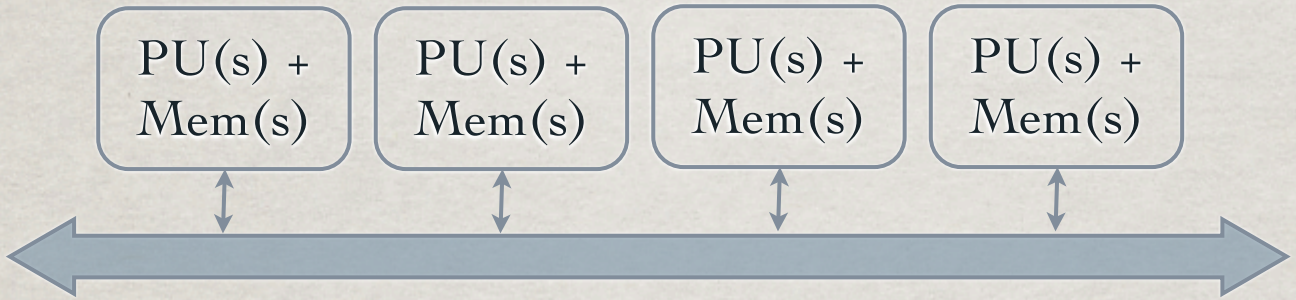
Regular interconnect: clusters, grids



Irregular interconnect: Internet, Clouds

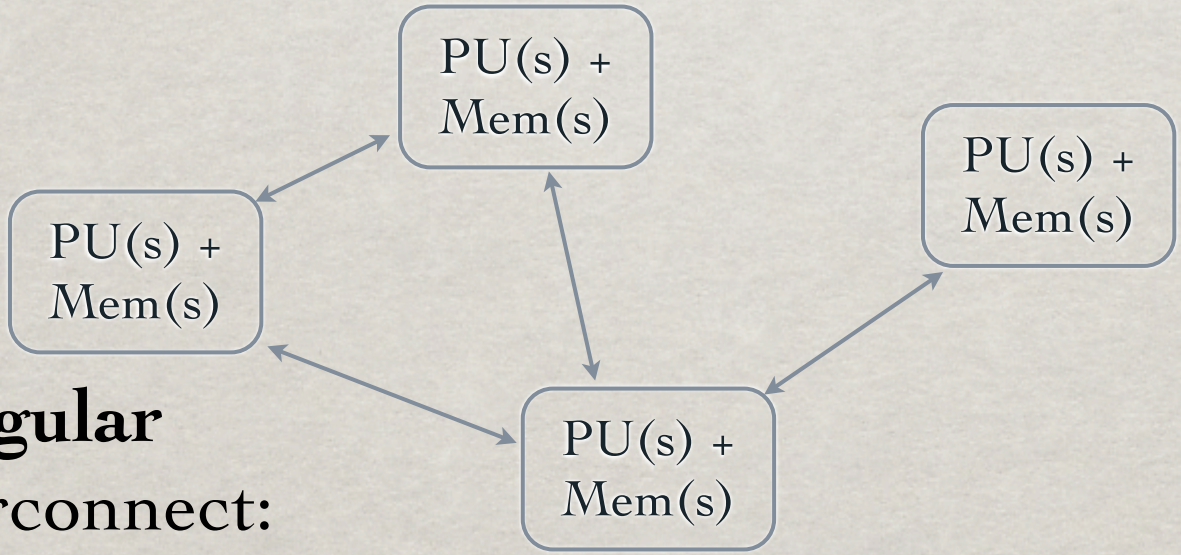
Topologies for distributed systems

"nodes"



Easier to program

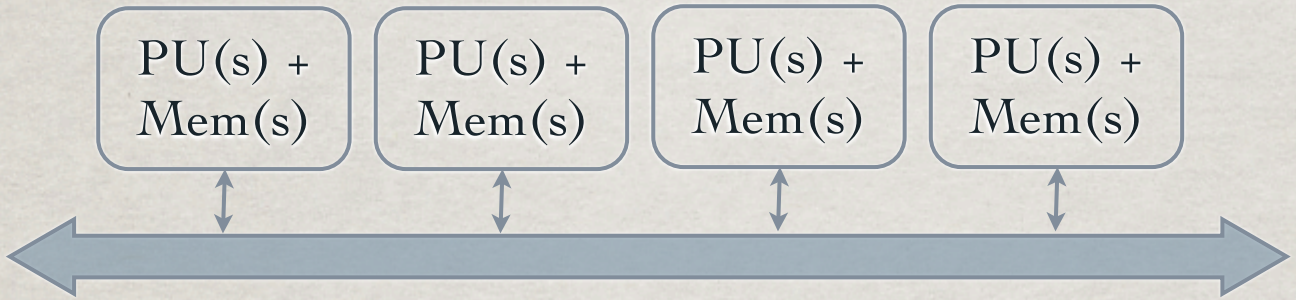
Regular interconnect: clusters, grids



Irregular interconnect: Internet, Clouds

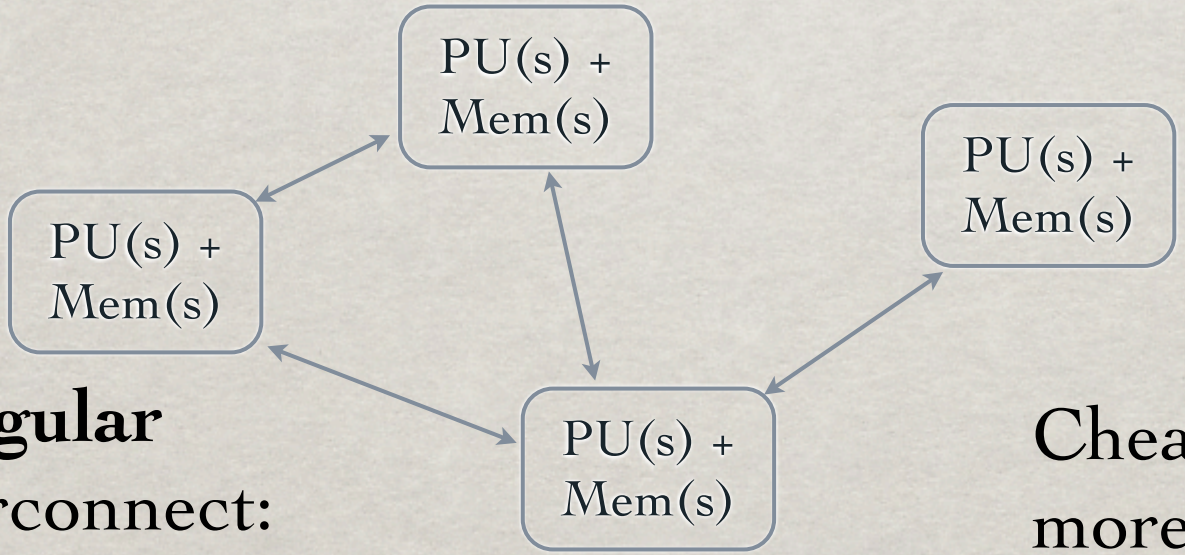
Topologies for distributed systems

"nodes"



Regular interconnect: clusters, grids

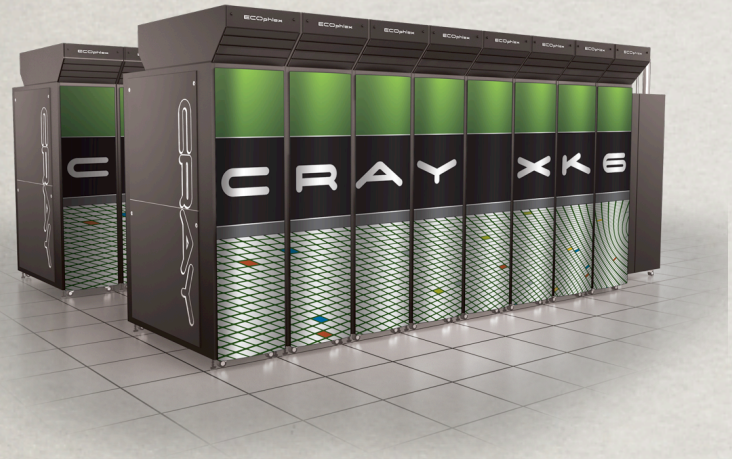
Easier to program



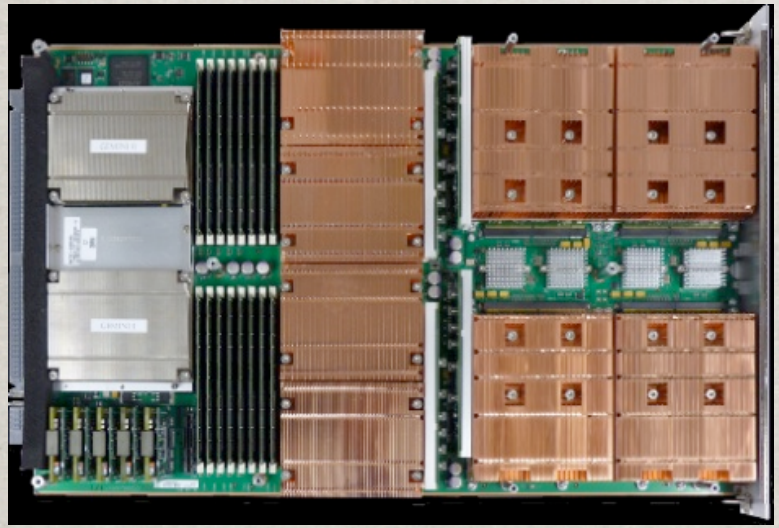
Irregular interconnect: Internet, Clouds

Cheaper, more robust to faults

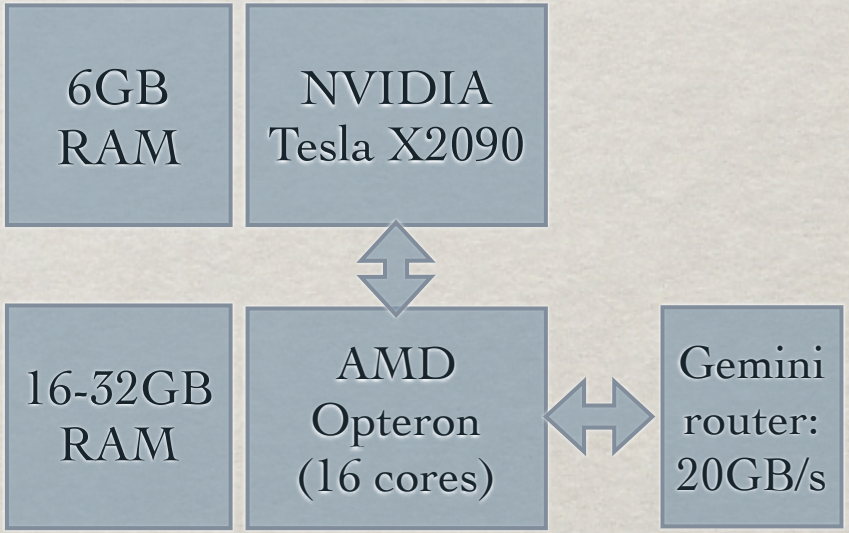
Examples



Cray XE6:
- 24 blades per cabinet
- 4 nodes per blade
- 16 cores + GPU per node
- 3D regular torus interconnect



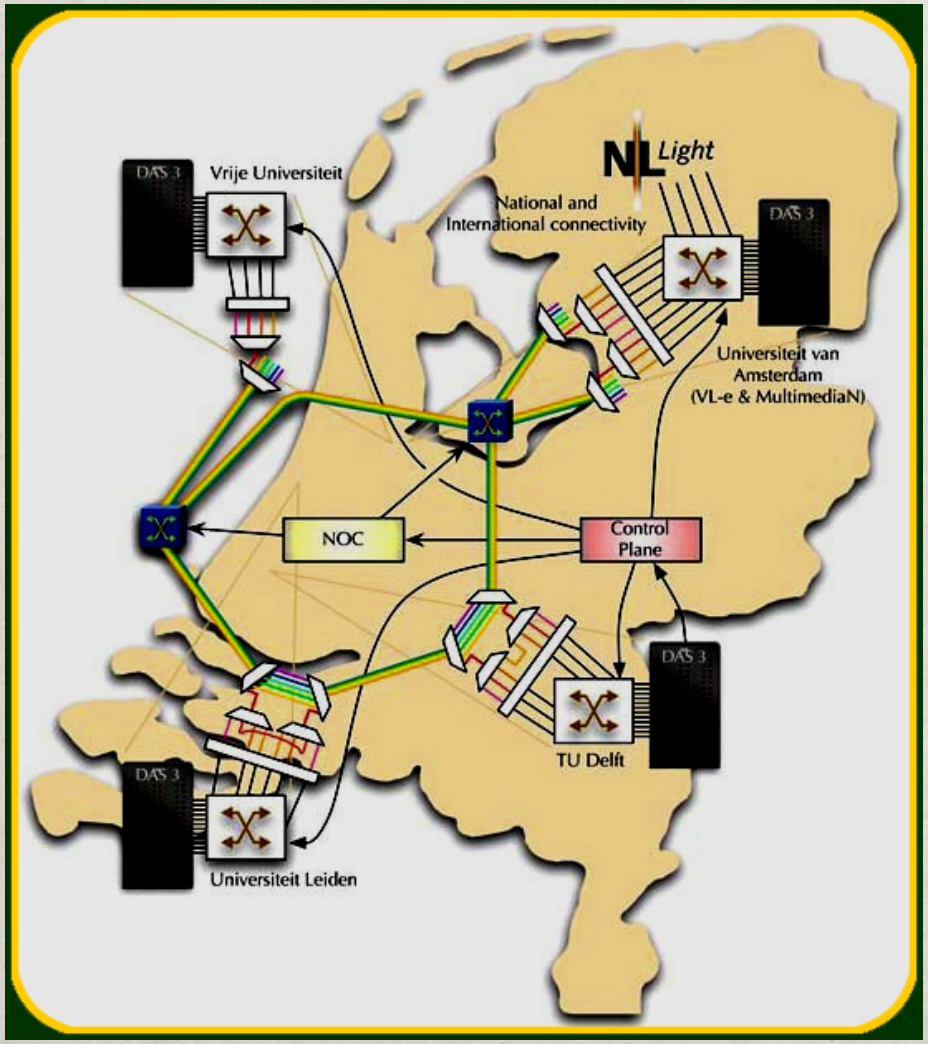
XE6 blade



XE6 node

inside bandwidth: 52GB/s

Examples



DAS-4:
- 4 sites
- 16-74 nodes per site
- 8 cores + accels per node
- 24-48GB RAM per node
- GbE network for control
- InfiniBand network for throughput: 20GB/s

DAS: Distributed ASCI Computer

Summary

Summary

- Characteristics of a distributed system?

Summary

- Characteristics of a distributed system?
  - Two levels: “local” (nodes) and “network”

Summary

- Characteristics of a distributed system?
  - Two levels: “local” (nodes) and “network”
  - Lots (GBs/TBs) of local memory/storage per node

Summary

- Characteristics of a distributed system?
  - Two levels: “local” (nodes) and “network”
  - Lots (GBs/TBs) of local memory/storage per node
  - Access times: fast to local node, longer to other nodes, typically 2x-100x difference

Summary

- Characteristics of a distributed system?
  - Two levels: “local” (nodes) and “network”
  - Lots (GBs/TBs) of local memory/storage per node
  - Access times: fast to local node, longer to other nodes, typically 2x-100x difference
  - Variable size by adding/removing nodes

Part 2: basics of Parallel programming

Essence of parallel programming

- Maximize throughput : computations / second
- Requires to **decompose problems** into separate sub-computations
- Two separate **phases**:
  - **Exposing concurrency**:
    find and describe the sub-problems
    (done by the human)
  - **Parallel execution**:
    distribute the sub-problems on the platform
    (shared responsibility between human and computer)

Example: data parallelism

- Multiply two vectors:

```
S = A[0]*B[0] + A[1]*B[1] + ... + A[n]*B[n]
```

- Sequential program:

```
# Python
s = 0
for i in range(n):
    s += a[i] * b[i]
```

```
/* C */
s = 0;
for (i = 0; i < n; ++i)
    s += a[i] * b[i];
```

- Can we not do better?

Example: data parallelism

- Decomposition: each A[i]*B[i] can be **computed independently**
- Solution:
  - decompose in **p** partial sums; each partial sum **k** has **n/p** steps to compute:

```
def partialsum(k, p, n, A, B):
    s = 0
    for i in range(k * n/p, (k + 1) * n/p):
        s = A[i] * B[i]
    return s
```

  - all partial sums can execute in parallel; the partial results can be summed at the end

Granularity

Granularity

- n = problem size

Granularity

- n = problem size
- p = number of independent sub-computations

Granularity

- n = problem size
- p = number of independent sub-computations
- n/p = "how much work to do per sub-computation"
  = **granularity**

Granularity

- n = problem size
- p = number of independent sub-computations
- n/p = "how much work to do per sub-computation"
  = **granularity**
- Trade offs?

Granularity

- n = problem size
- p = number of independent sub-computations
- n/p = "how much work to do per sub-computation"
  = **granularity**
- Trade offs?
  - **p small: less concurrency**
    less opportunity for speed-up, **lower throughput**

Granularity

- n = problem size
- p = number of independent sub-computations
- n/p = "how much work to do per sub-computation"
  = **granularity**
- Trade offs?
  - **p small: less concurrency**
    less opportunity for speed-up, **lower throughput**
  - **p large: more communication and synchronization** may introduce overhead that **limit performance**

Granularity

- n = problem size
- p = number of independent sub-computations
- n/p = "how much work to do per sub-computation"
  = **granularity**
- Trade offs?
  - **p small: less concurrency**
    less opportunity for speed-up, **lower throughput**
  - **p large: more communication and synchronization** may introduce overhead that **limit performance**
- In practice: choose **p** to match the number of PUs

Data parallelism: SIMD / SPMD

Data parallelism: SIMD / SPMD

- What defines data parallelism?

Data parallelism: SIMD / SPMD

- What defines data parallelism?
  - The same function (instruction/program) is applied to each item in the input data: SI/SP

Data parallelism: SIMD / SPMD

- What defines data parallelism?
  - The same function (instruction/program) is applied to each item in the input data: SI/SP
  - The large number of items in the input data provides concurrency: MD

Data parallelism: SIMD / SPMD

- What defines data parallelism?
  - The same function (instruction/program) is applied to each item in the input data: SI/SP
  - The large number of items in the input data provides concurrency: MD
  - General form: `for i: Output[i] = f(Input[i])`
    Also noted: `Out = parallel_map(n, p, f, Input)`

Reductions

- Typically a data parallel computation is **followed by a reduction**
- In the previous example:

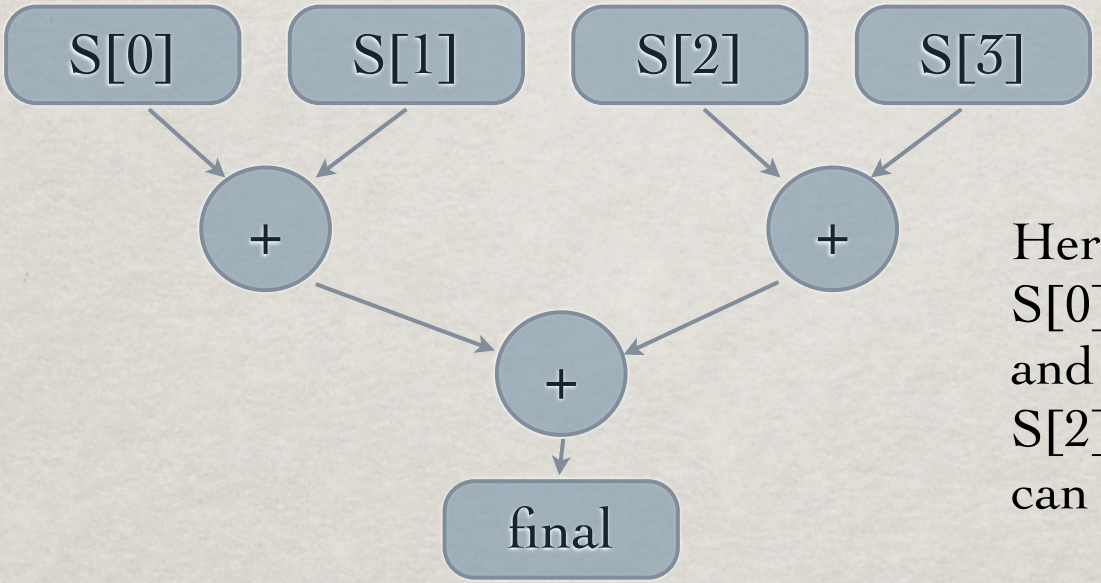
```
S = parallel_map(n, p, partialsum, (A, B))

final = 0
for k in range(p): # reduce
    final += S[k]
```

- General form of a reduction:

$$X = A[0] \odot A[1] \odot \ldots \odot A[n]$$

Reductions

- Reductions can be partially parallelized too!



Here
S[0]+S[1]
and
S[2]+S[3]
can be computed in parallel

- Condition: the operation must be **associative** and **commutative,** eg: min, max, +, *, and, or

- Noted also: `Out = parallel_reduce(n, p, op, Input)`

Example map/reduce (simplified)

Example map/reduce (simplified)

- Sequential:

```
def prod(n, A, B):
  sum = 0
  for i in range(n):
     sum += A[i] * B[i]
  return sum
```

Example map/reduce (simplified)

- Sequential:

```
def prod(n, A, B):
  sum = 0
  for i in range(n):
     sum += A[i] * B[i]
  return sum
```

- Parallel:

```
def parallel_prod(n, A, B, p):
  return parallel_reduce(n, p, __add__,
              parallel_map(n, p, partialsum, (A, B))
```

Example map/reduce (simplified)

- Sequential:

```
def prod(n, A, B):
  sum = 0
  for i in range(n):
     sum += A[i] * B[i]
  return sum
```

- Parallel:

```
def parallel_prod(n, A, B, p):
  return parallel_reduce(n, p, __add__,
                parallel_map(n, p, partialsum, (A, B))
```

- How many steps? eg with n=10^6 and p=1000

Example map/reduce (simplified)

- Sequential:

```
def prod(n, A, B):
  sum = 0
  for i in range(n):
     sum += A[i] * B[i]
  return sum
```

- Parallel:

```
def parallel_prod(n, A, B, p):
  return parallel_reduce(n, p, __add__,
                parallel_map(n, p, partialsum, (A, B))
```

- How many steps? eg with n=10^6 and p=1000
  - Sequential: **n steps**, eg 10^6 steps

Example map/reduce (simplified)

- Sequential:

```
def prod(n, A, B):
  sum = 0
  for i in range(n):
     sum += A[i] * B[i]
  return sum
```

- Parallel:

```
def parallel_prod(n, A, B, p):
  return parallel_reduce(n, p, __add__,
                parallel_map(n, p, partialsum, (A, B))
```

- How many steps? eg with n=10^6 and p=1000
  - Sequential: **n steps**, eg 10^6 steps
  - **p** parallel PUs:
    n / p steps for map + **log(p) steps** for reduce = 1006 steps

Exposing concurrency with data parallelism

- **Role of the programmer:**
  - **Find** and **describe** the **independent sub-problems**
  - **Describe the dependencies** for reductions
  - If possible, leave the parameter **p configurable** at run-time
- The **programming environment** helps:
  - Usually provides an **API for map and reduce** (no need to manually code the distribution and communication)
  - Sometimes also determines **p** automatically

Summary

- You: **decompose** the problem, try to express using **n/p complexity**; computer: run work over parallel resources
- Limits: p too small = less throughput, p too big = more overhead; try p ≈ number of PUs
- **map/reduce** suitable for **data-parallelism followed by reductions**

Part 3: patterns of distributed programming

Limits of data-parallelism

Limits of data-parallelism

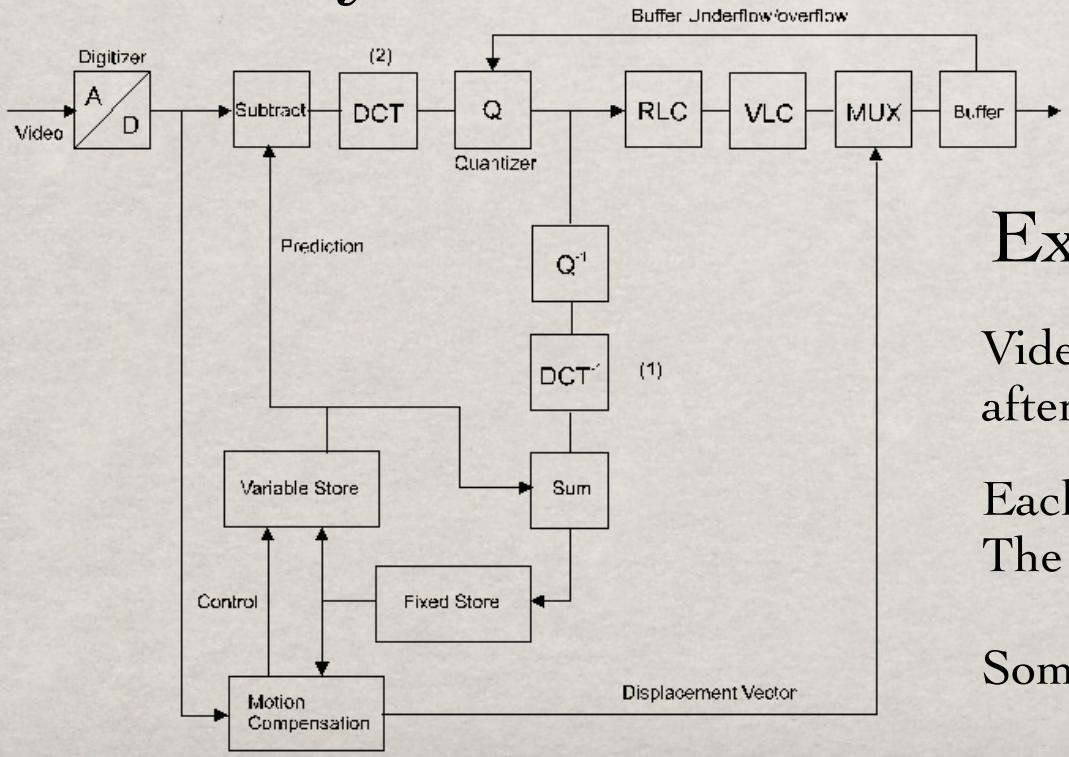
- Unfortunately not all problems are purely data-parallel

Limits of data-parallelism

- Unfortunately not all problems are purely data-parallel
- Can you think of counter examples?

Limits of data-parallelism

- Unfortunately not all problems are purely data-parallel
- Can you think of counter examples?



Example: MPEG encode

Video stream: many frames(picture) one after another

Each frame decomposed in blocks
The workload per frame varies

Some dependencies across frames

Limits of data-parallelism

Limits of data-parallelism

- Also, SIMD/SPMD case yields best throughput
  only if all p PUs run at equal speed

Limits of data-parallelism

- Also, SIMD/SPMD case yields best throughput
  only if all p PUs run at equal speed

- Is this always guaranteed?

Limits of data-parallelism

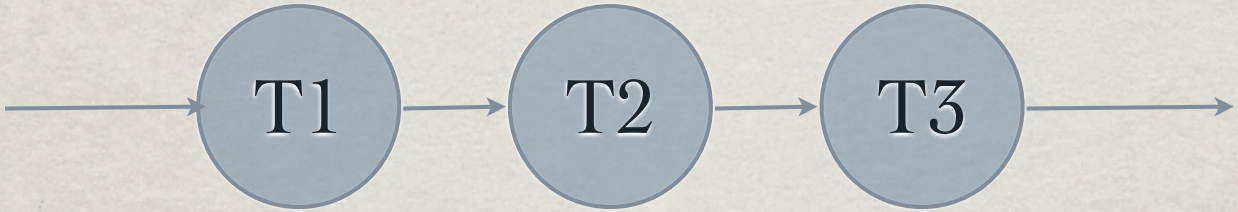
- Also, SIMD/SPMD case yields best throughput
  only if all p PUs run at equal speed
- Is this always guaranteed?
  - heat / cooling induces **transient failures/ slowdowns**

Limits of data-parallelism

- Also, SIMD/SPMD case yields best throughput
  only if all p PUs run at equal speed
- Is this always guaranteed?
  - heat / cooling induces **transient failures/ slowdowns**
  - **interference from other computations** running on the same resoures

Other patterns: pipeline

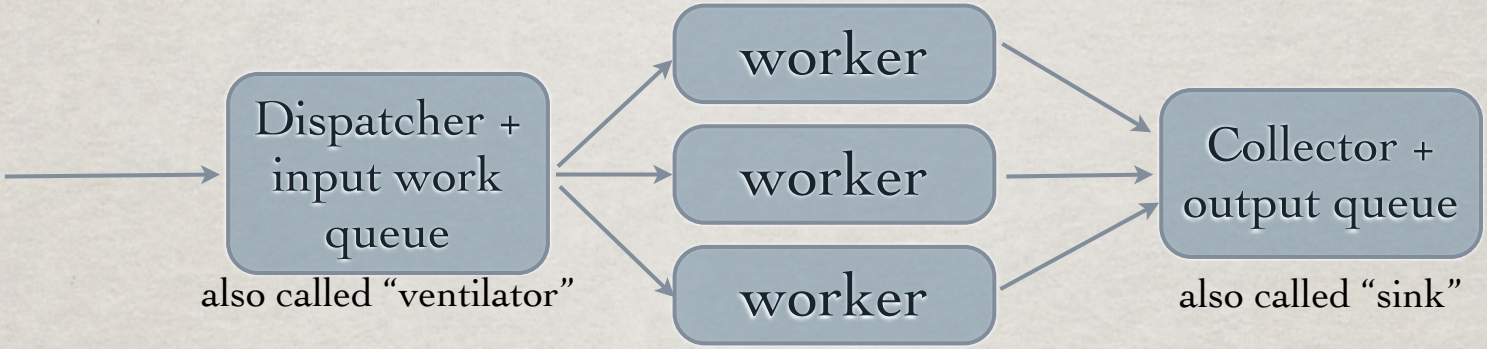
- concept: “assembly line”



- Works best when each sub-task takes approximately the same time
- Typical example: signal processing, video

Other patterns: Farming

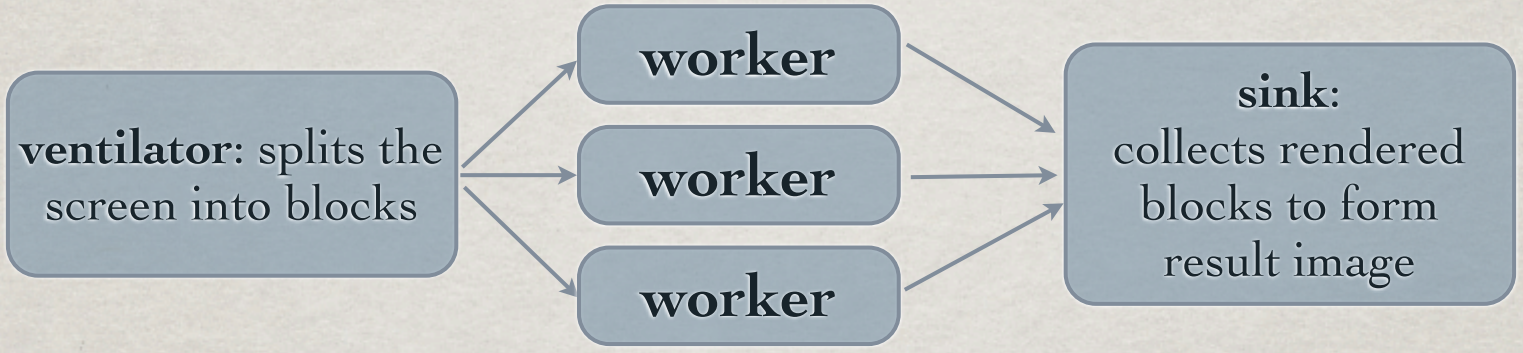
- concept: “clerks at the city hall”



- Works best when each task work is independent from the others + separate data
- (Can be used to simulate SIMD/SPMD on heterogeneous platforms)
- Typical example: web server, database

Practicals

Example: raytracer



- We provide you Python source code for the 3 components
- It uses ∅MQ for communication: a library for inter-process messaging
- Your task: try it out, optimize: how many workers and which block size are best?
- Run on your local workstation / local network

Basics with MPI

- MPI is an "old school" API for parallel programming - more mature, widely used
- Like ØMQ, MPI provides send/recv
- MPI also provides automated reduction, barriers and other abstractions to simplify the programmer's job

Basics with MPI

- Basic principle: MPI automatically starts **the same C program multiple times** on multiple processes/nodes

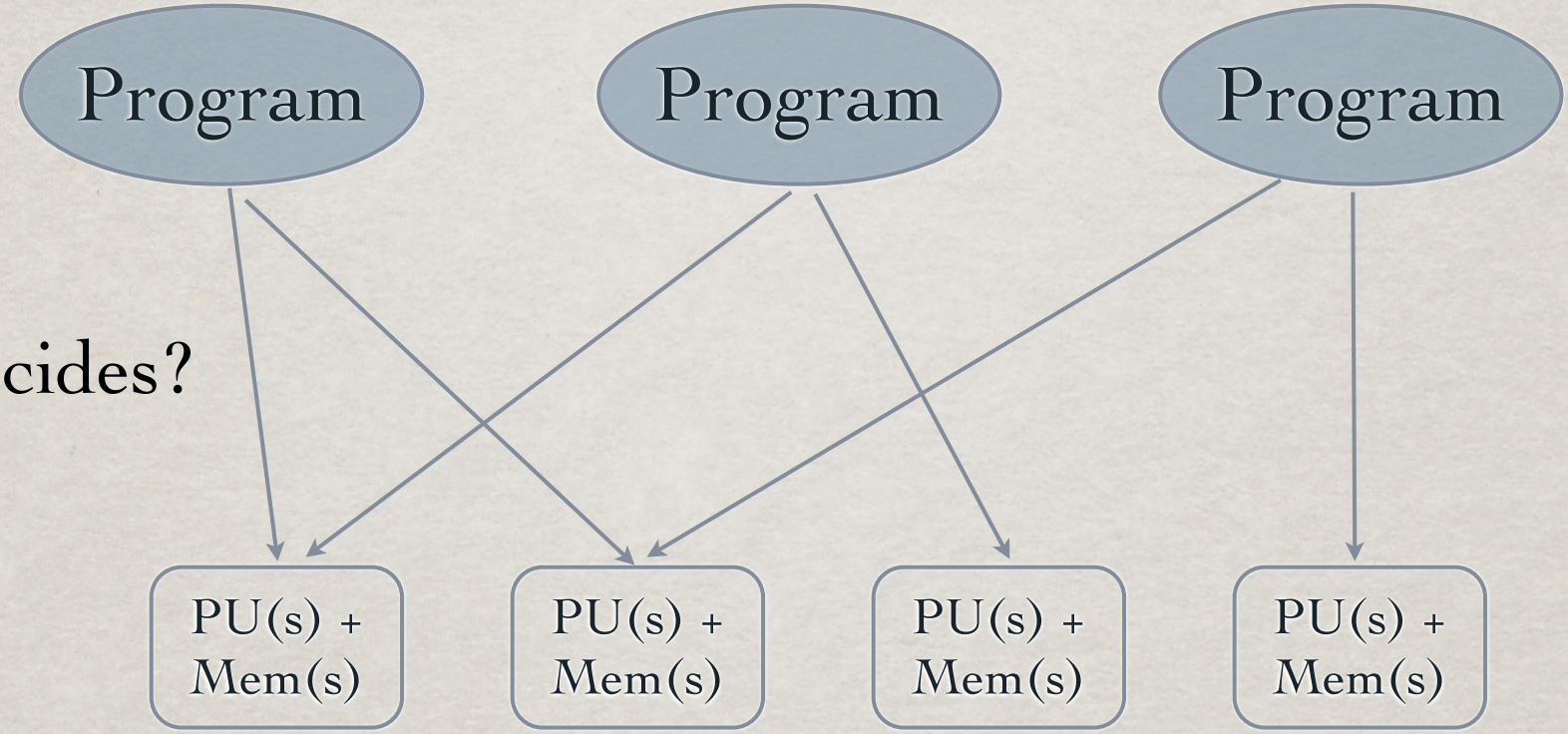
- Example:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv){
  int k, p;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &k);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  printf("hello world k=%d p=%d\n", k, p);
  // (place computation here)
  MPI_Finalize();
  return 0;
}
```

Basics with MPI

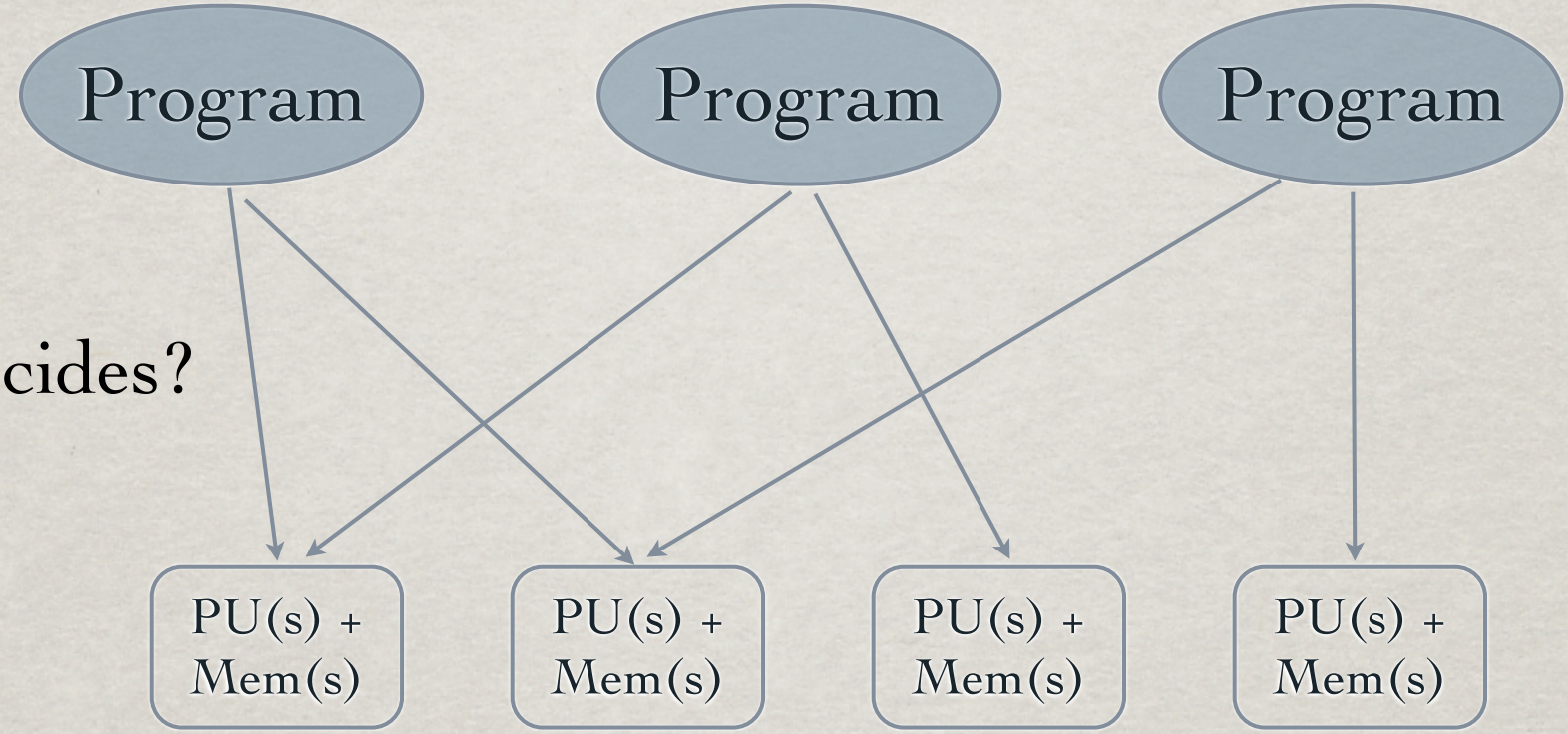
- SIMD/SPMD “map”: in MPI, no special primitive needed
  (the entire program is already duplicated)
- “reduce”: use MPI_Reduce(....)
- Check with the assistants for details!

Sharing resources



Who decides?

Sharing resources



Who decides?

Typically:

- programmers define **jobs**
- **job = program + resource request**
- a common **job manager** maps jobs to the compute nodes
- Example: http://www.cs.vu.nl/das4/jobs.shtml