Parallel overview

http://cees-gitlab.stanford.edu/bob/gp257_2020.git

Agenda

- Types of parallel computers
- Programing models
- Shared concepts
- Examples

Flynn's Taxonomy

SISD Single instruction, single data	SIMD Single instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

Flynn's Taxonomy

SISD Single instruction, single data	SIMD Single instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

load a(1)
c(1)=a(1)
store c(1)
load a(2)
c(2)=a(2)
store c(2)

Example: Most PCs
What we have talked about up to this point

Flynn's Taxonomy

SISD Single instruction, single data	SIMD Single instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

load a(1)
c(1)=a(1)
store c(1)

load a(2)
c(2)=a(2)
store c(2)

Example: Vector processors
Good for vector dominated codes

Flynn's Taxonomy

SISD Single instruction, single data	SIMD Single instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

load a(1)
c(1)=a(1)*a
store c(1)

load a(1)
c(2)=a(1)*b
store c(2)

Example: None*
Certain algorithms fit this model.

Flynn's Taxonomy

SISD Single instruction, single data	SIMD Single instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

load a(1)
c(1)=a(1)*a
store c(1)

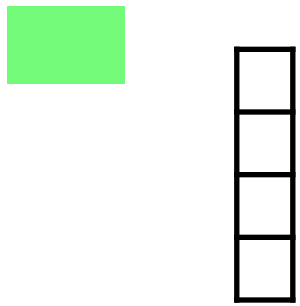
load a(2)
c(2)=a(2)*b
store c(2)

Example: Multi-core, SMP
What we will be discussing for the remainder of the class

Standard CPU: Threads

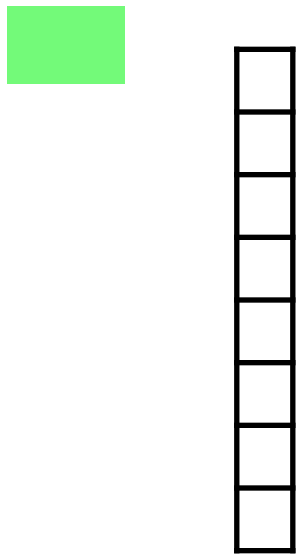
Thread contains a task or series of tasks you wish to do

Standard CPU: Vector



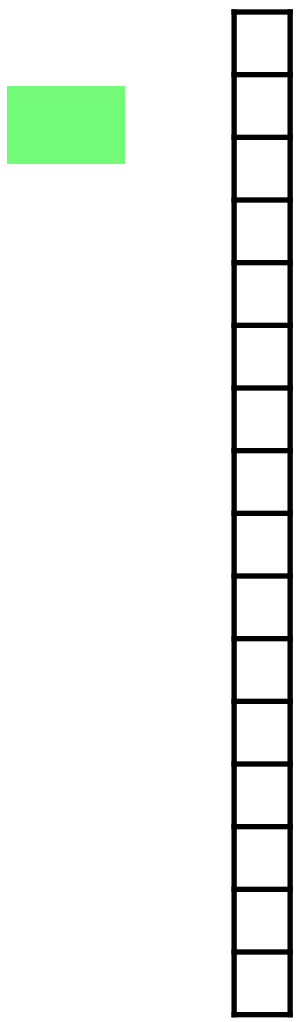
Pentium 3 (1999) allowed four simultaneous vector operations

Standard CPU: Vector



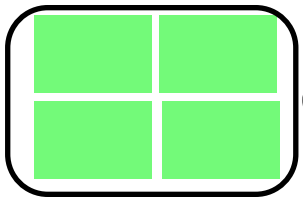
Sandy Bridge (2011)
increased vector length to 8

Standard CPU: Vector



Xeon Phi (2012) increased vector length to16

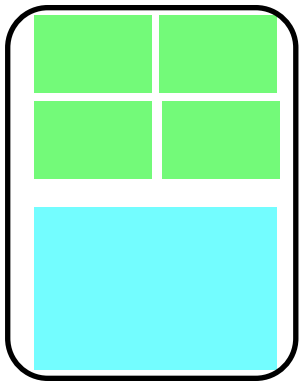
Standard CPU: Multiple threads per core



Core

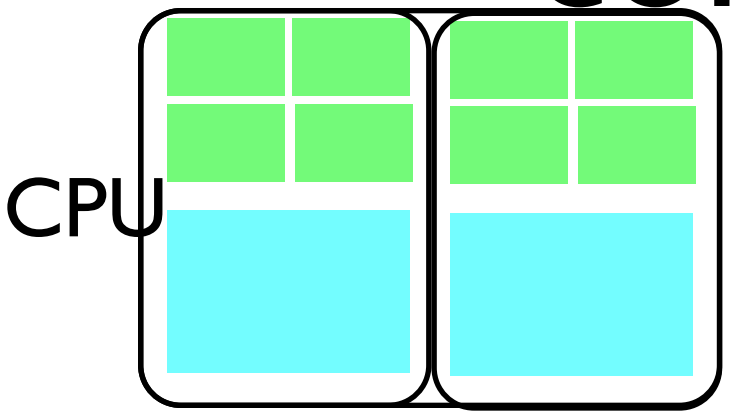
Multiple threads can exist on be targeted to a single core..we will come back to why

Standard CPU: Memory



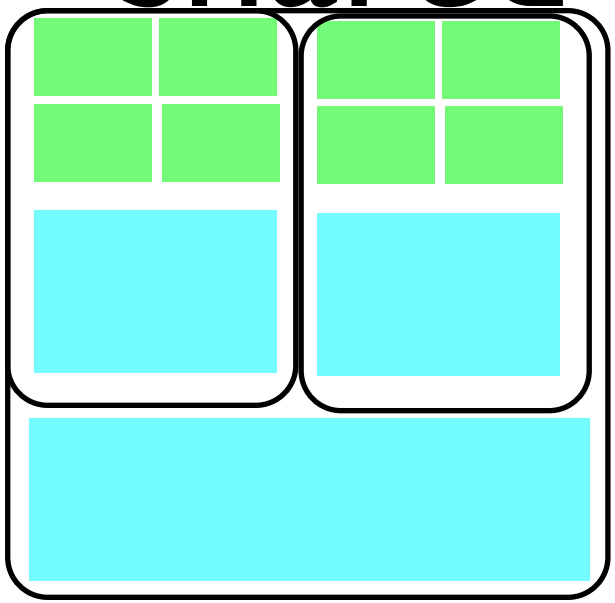
A core usually has its own memory (L1, L2)

Standard CPU: Multiple cores per CPU



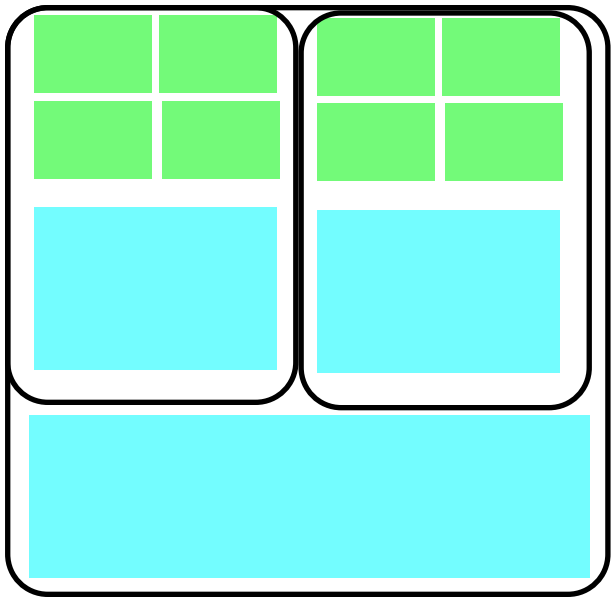
A single CPU is composed of multiple cores

Standard CPU: Memory shared on a single CPU



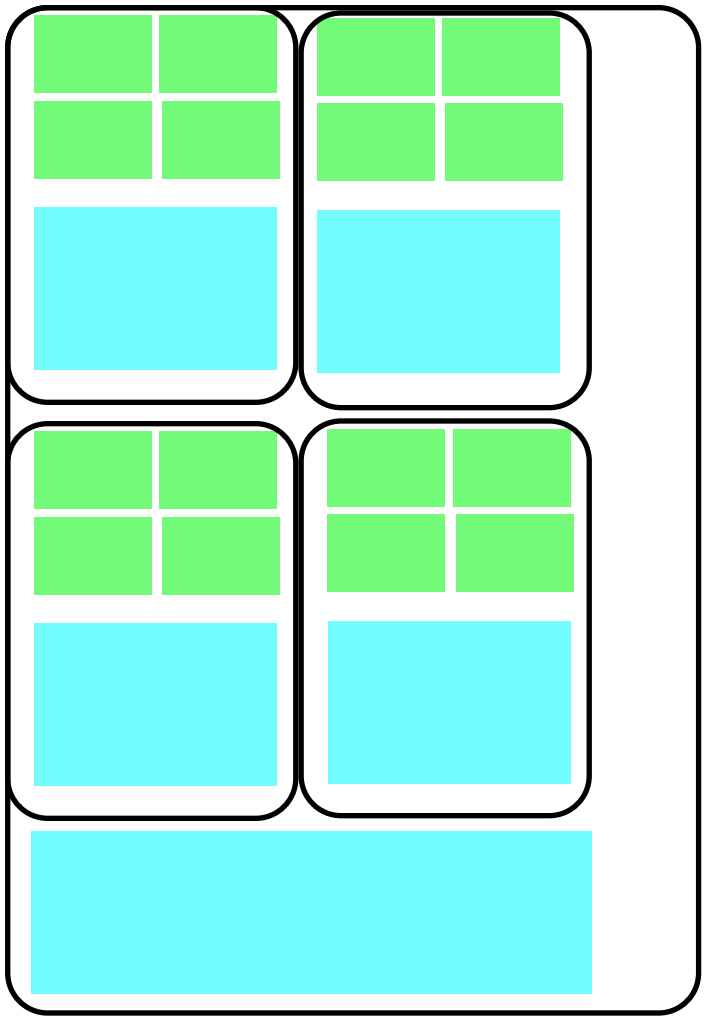
A single CPU often shares memory (L3)

Core growth



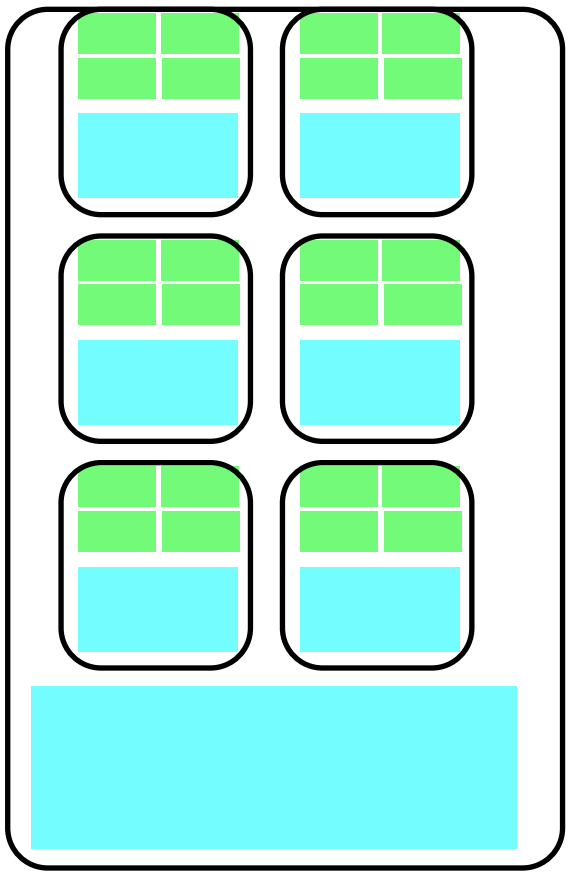
2004 Intel Pentium 4 with 2 cores

Core growth



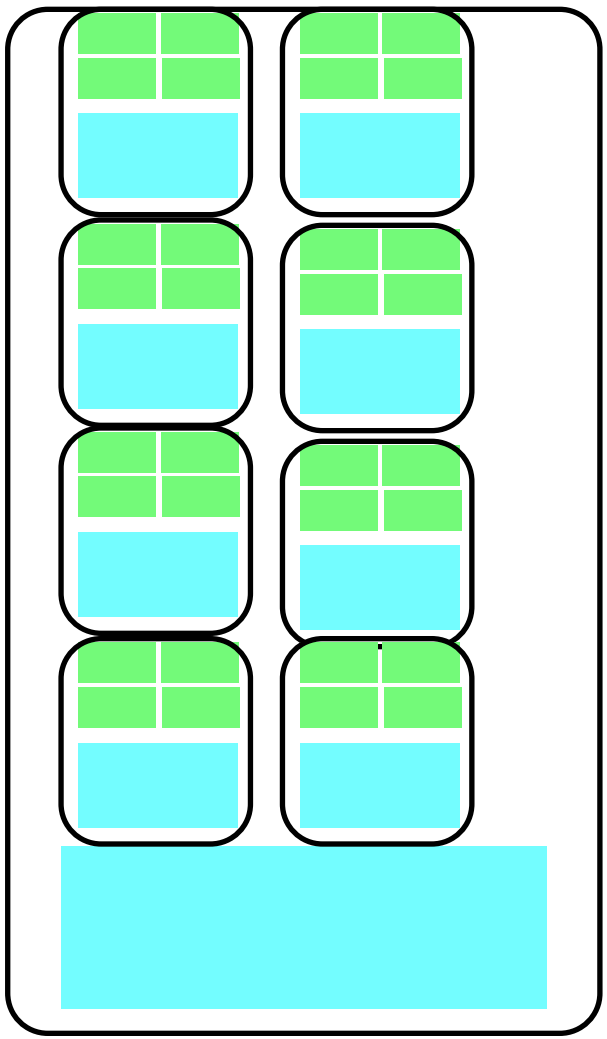
2006 Intel Xeon with 4 cores

Core growth



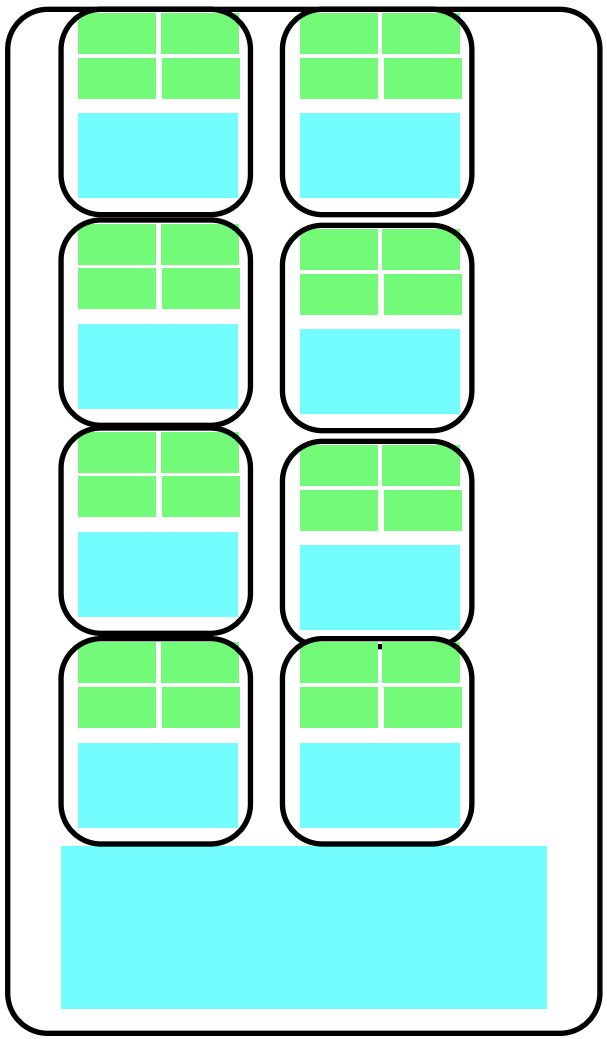
2009 Intel Woodcrest with 6 cores

Core growth



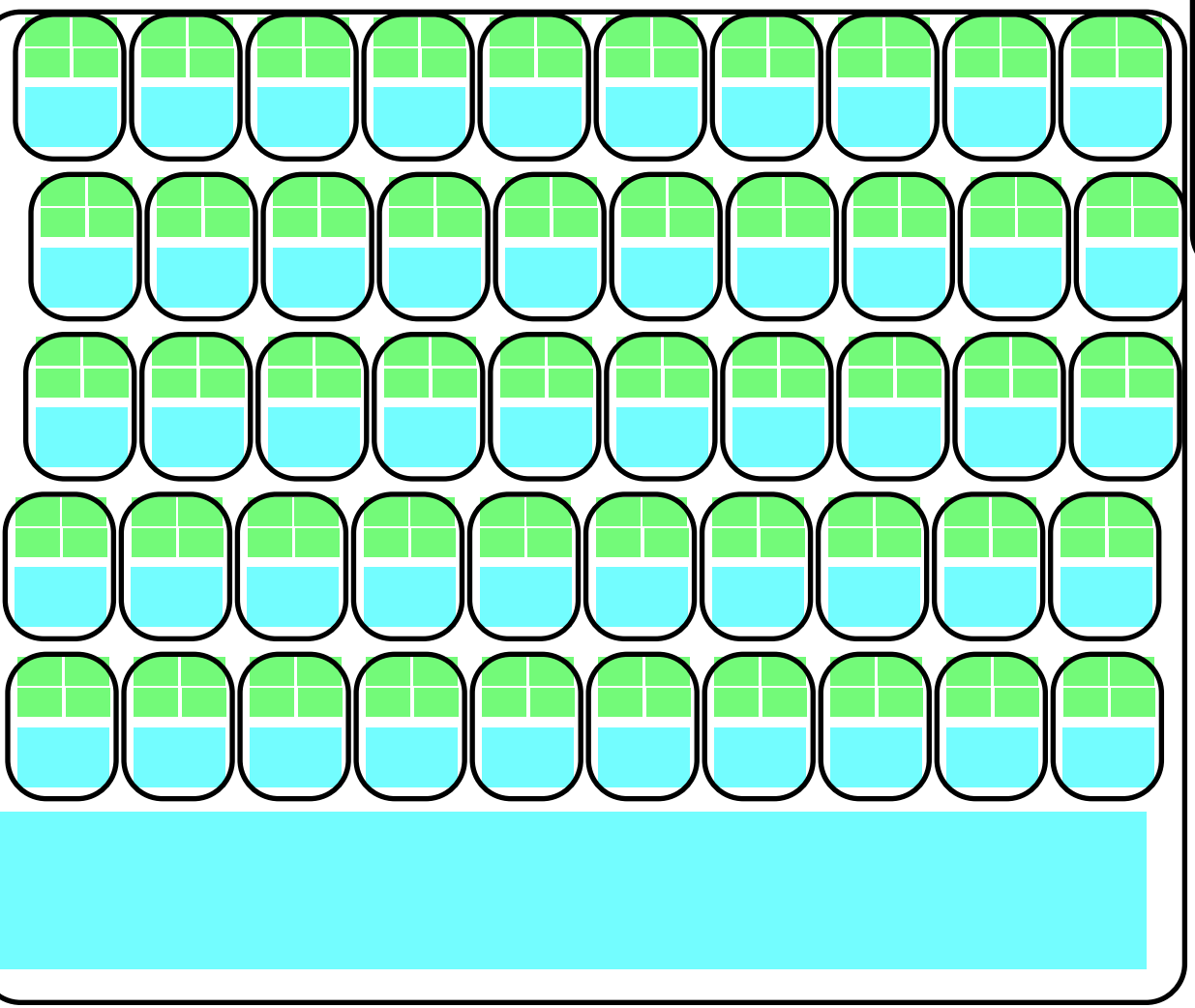
2011 Intel Sandy Bridge
with 8 cores

Core growth



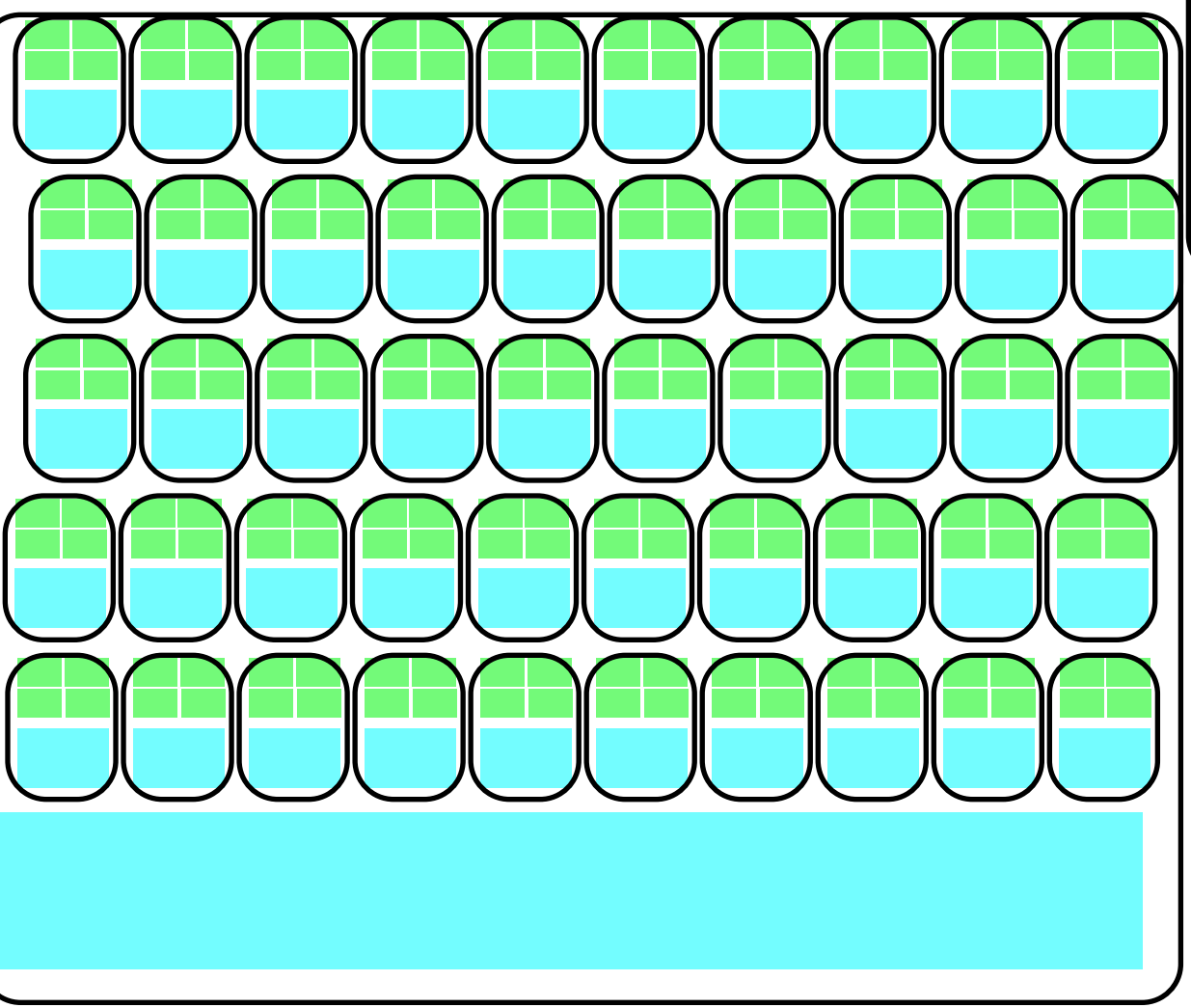
2020 Intel
Cascade lake
56

Core growth



2012 Intel Xeon Phi with 50 cores

Nvidia



2012 K20

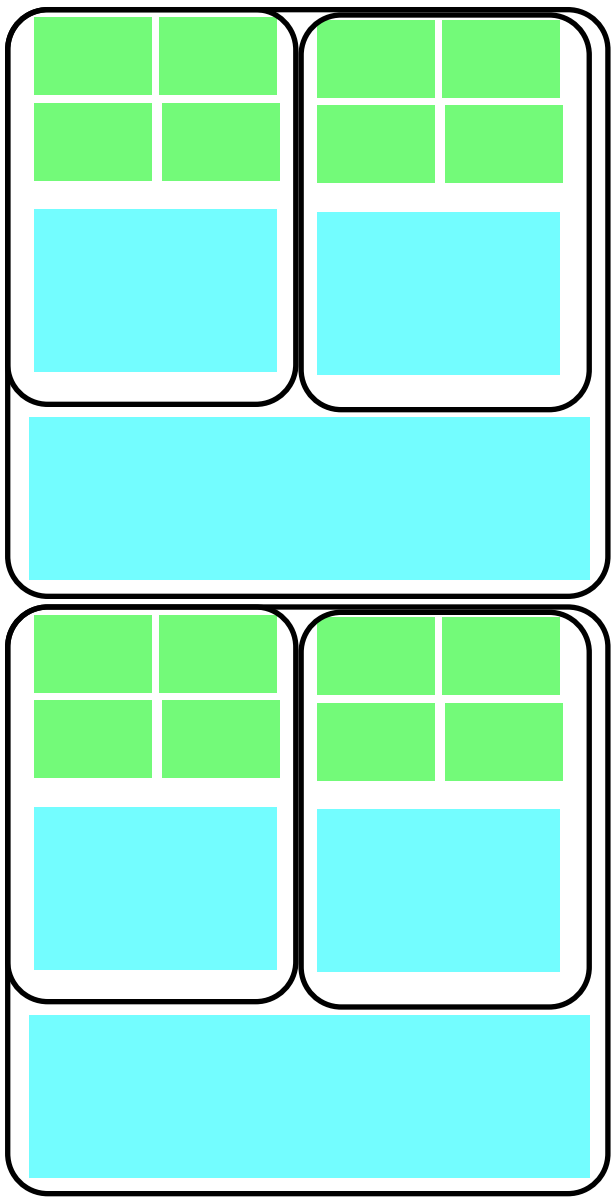
512 cores

Nvidia

2020 A100

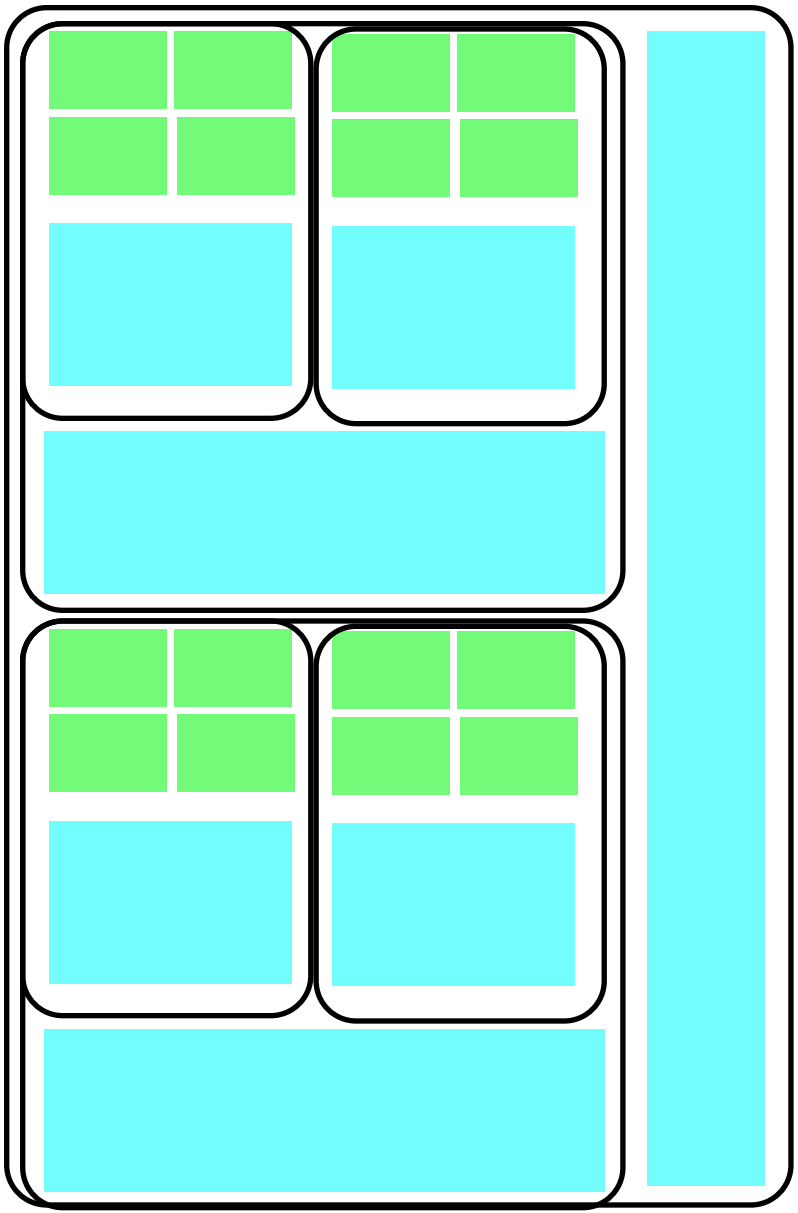
6000 cores

Multiple processors per board



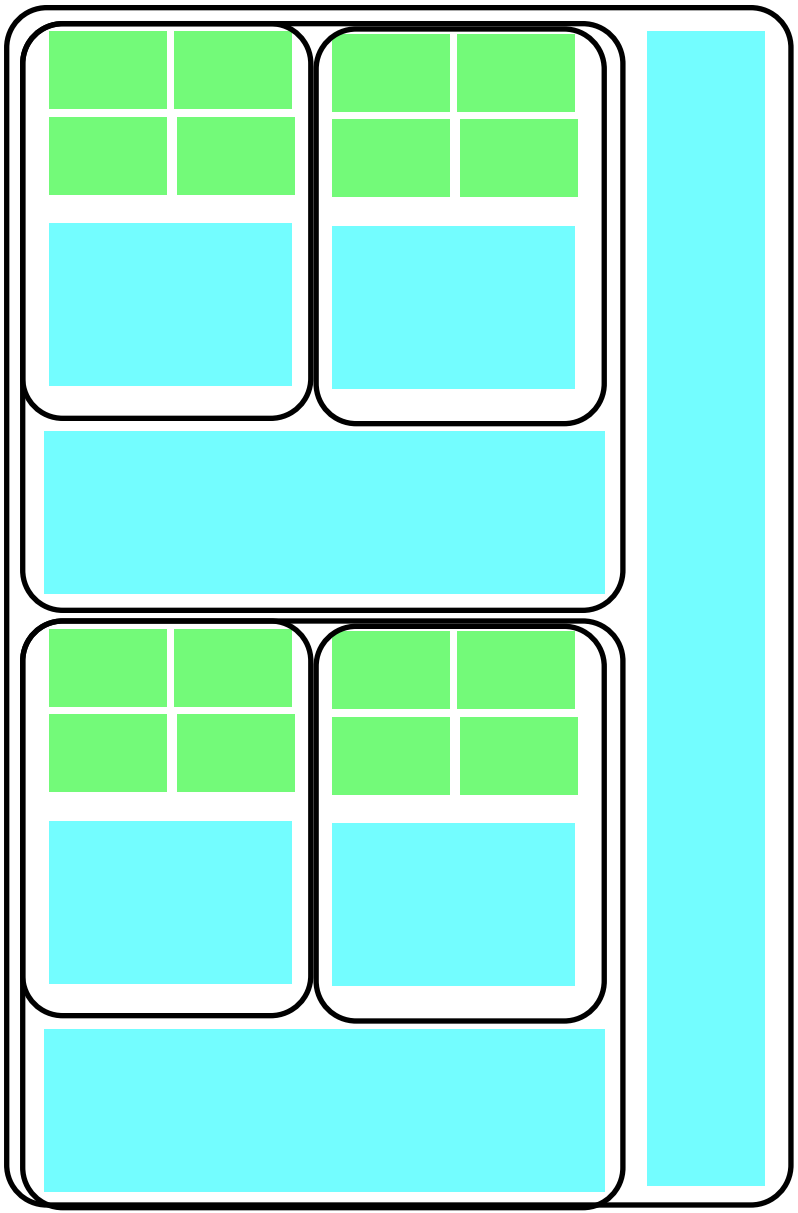
Multiple CPUs per motherboard

Multiple processors per board



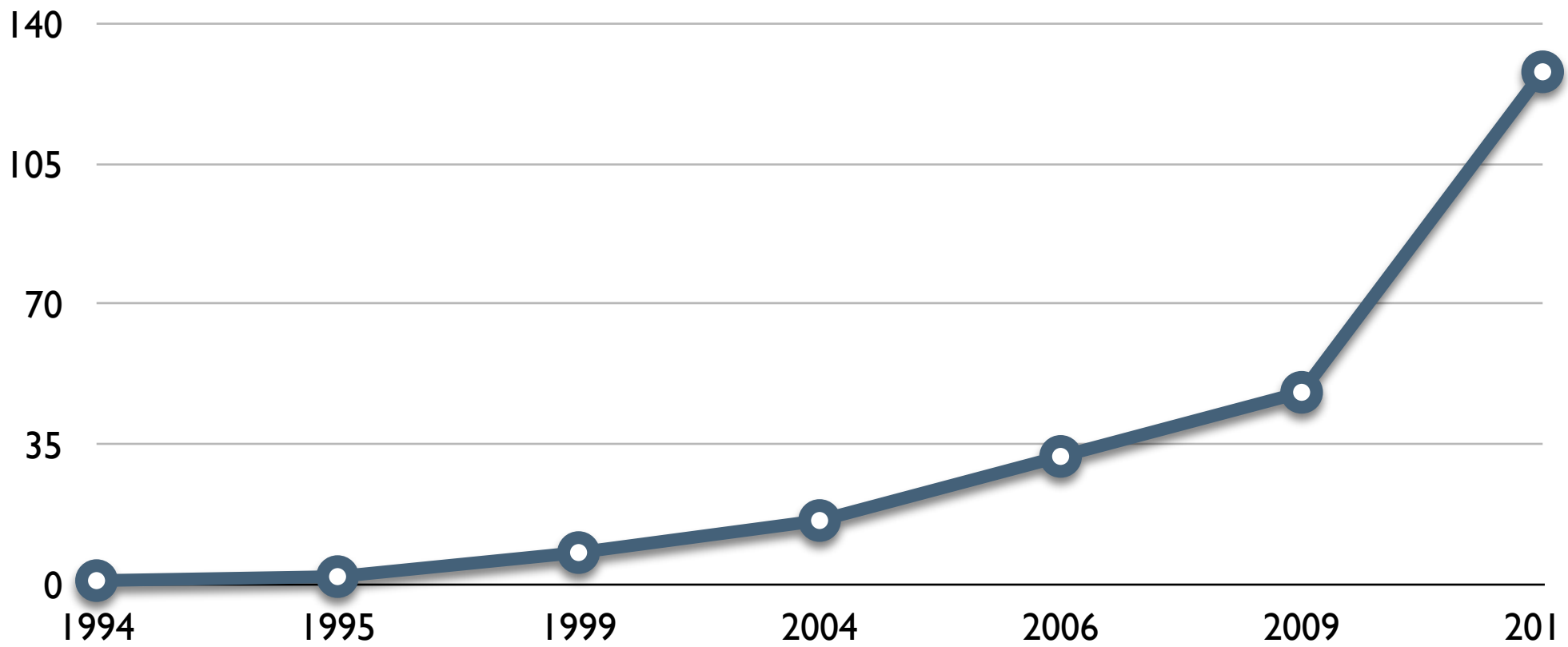
Shared memory on a motherboard

Multiple processors per board

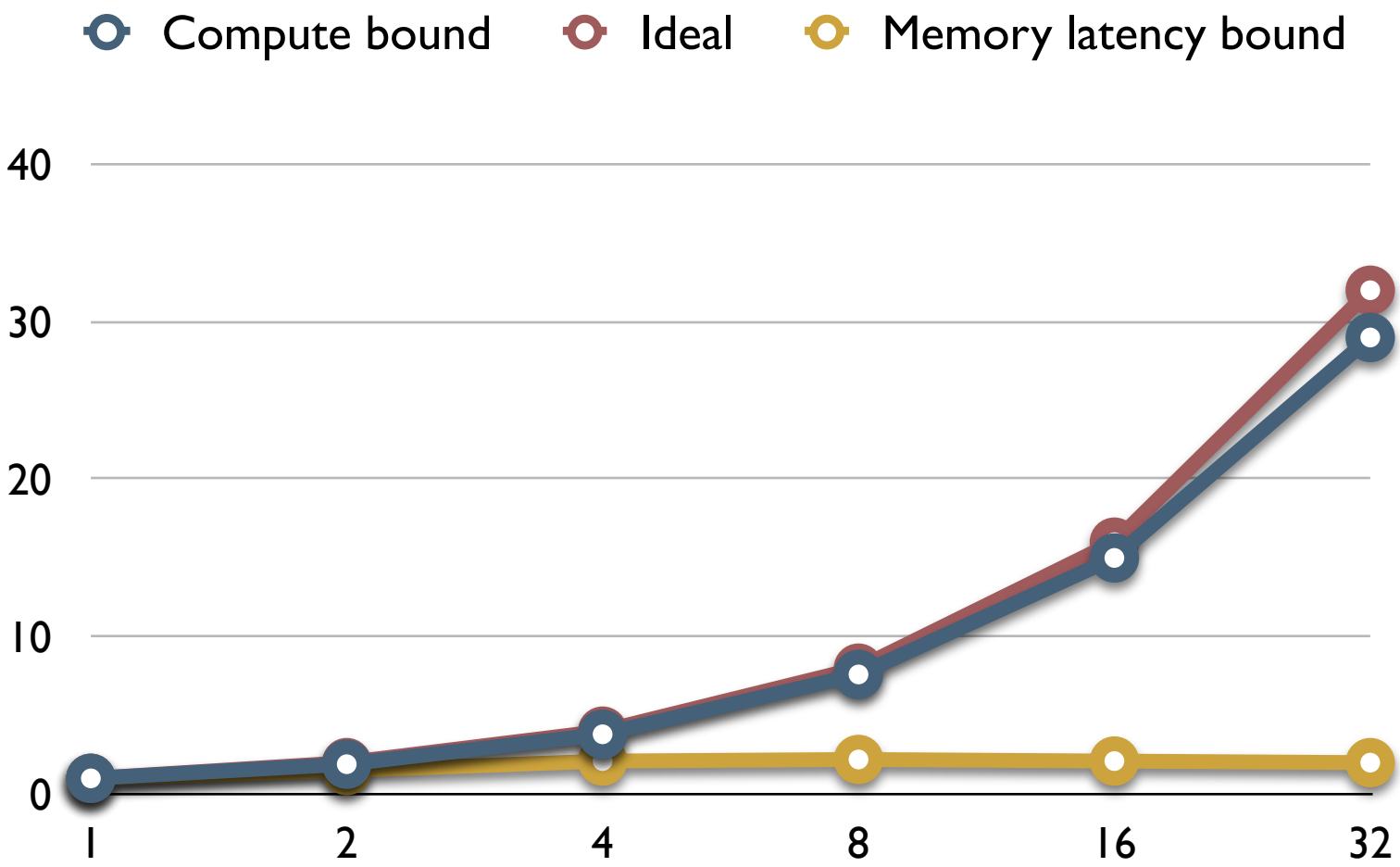


Shared memory on a motherboard

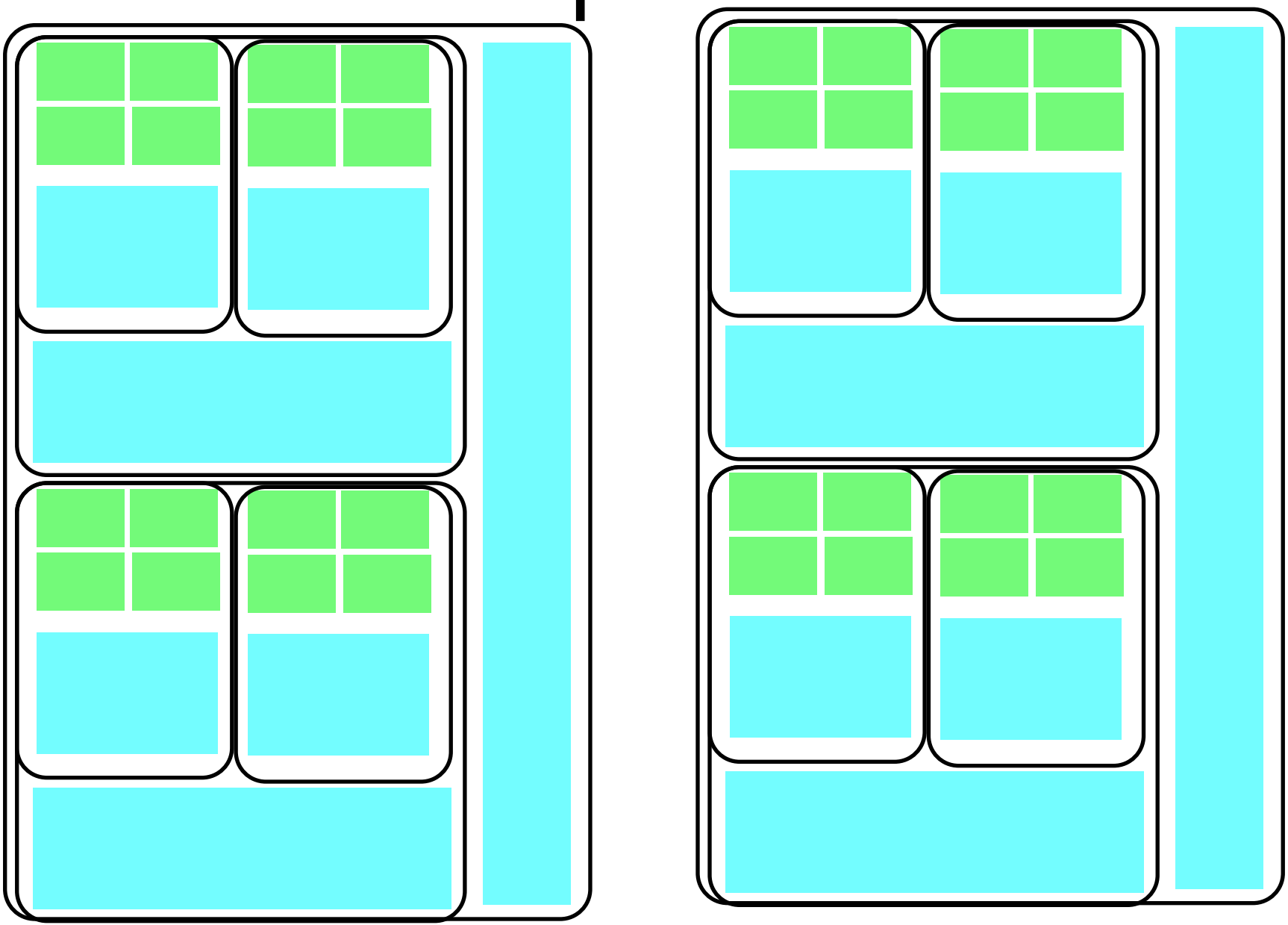
Parallel growth (conventional Intel)



Compute vs. memory bound



Cluster made up of multiple computer units



Parallel computer memory architectures

- Shared memory architectures - all processors have access to all memory

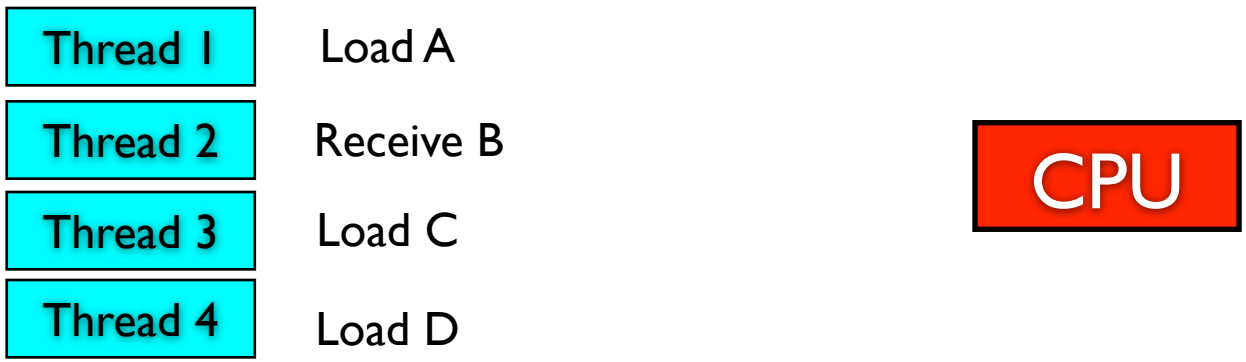
- Distributed memory architectures - programer manages communicating information between nodes

Chip-level multiprocessing (CMP)

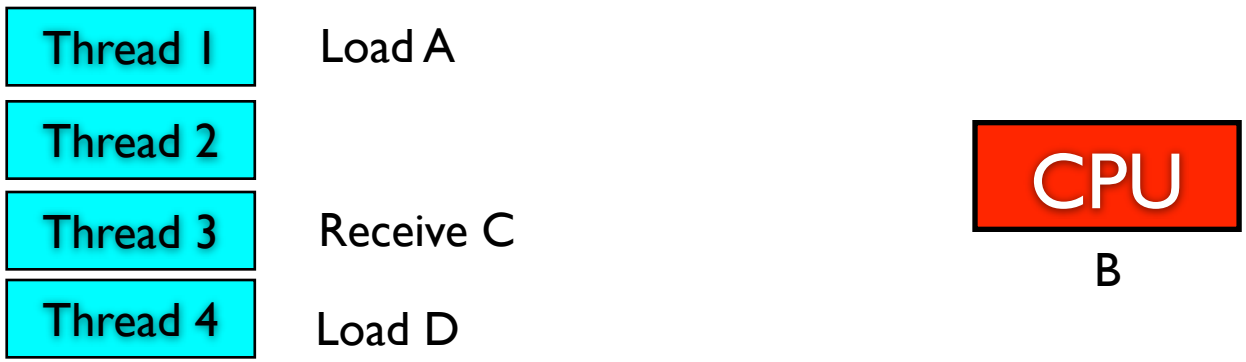
- Multiple cores that exist on a single chip
- Share memory at some level
- Large speedup for compute bound applications

Simultaneous multithreading (SMT)

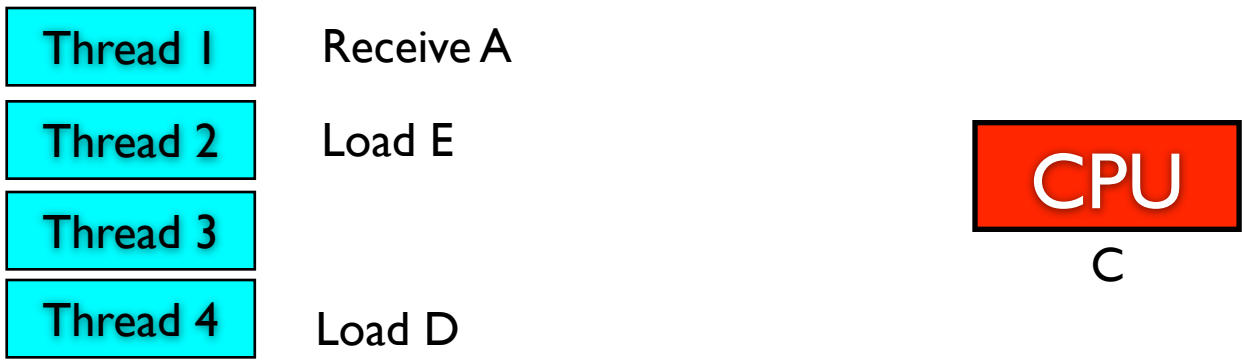
Simultaneous multithreading (SMT)



Simultaneous multithreading (SMT)

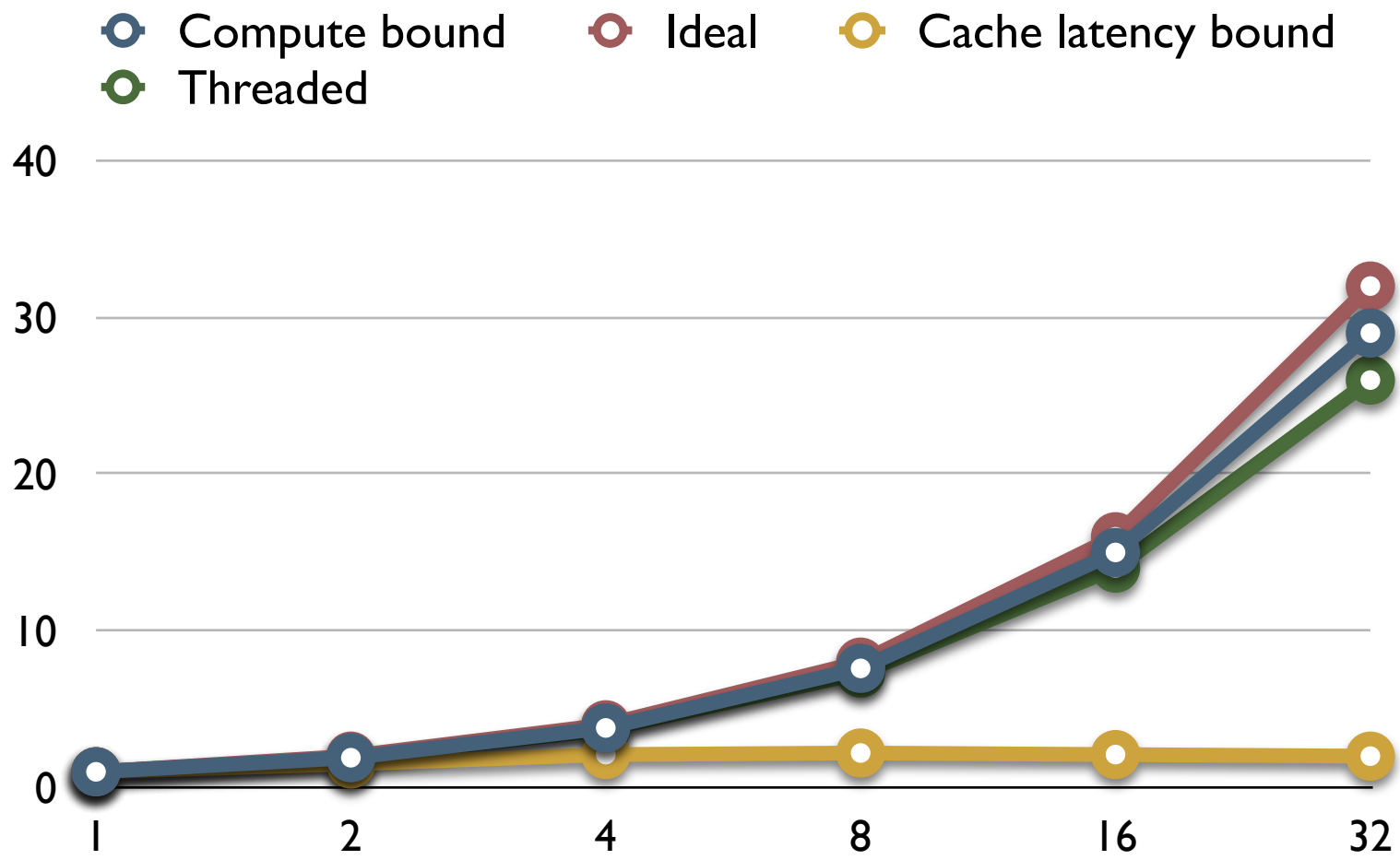


Simultaneous multithreading (SMT)

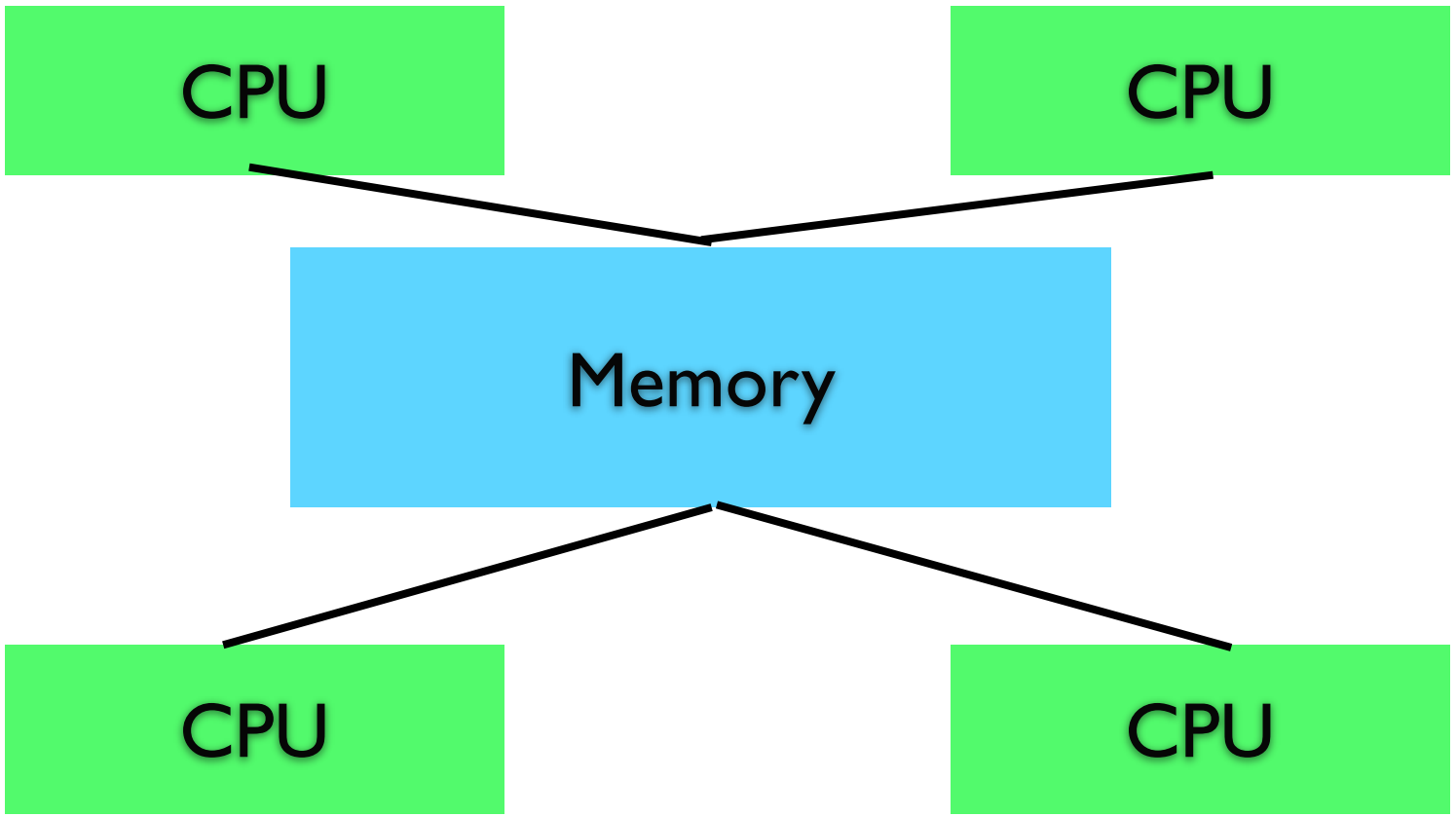


Sun Niagra & Niagra II, GPU

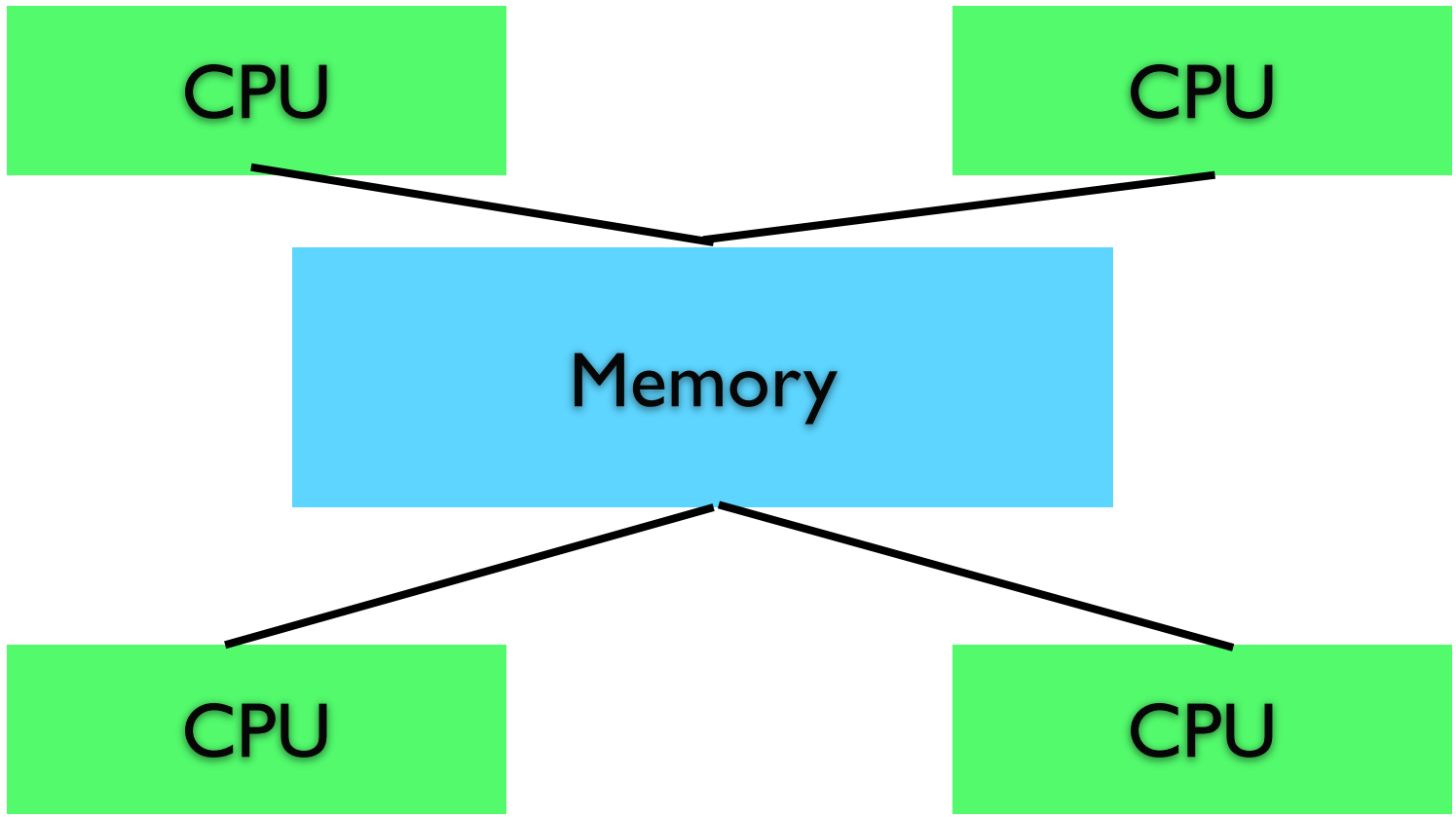
Compute vs. memory bound



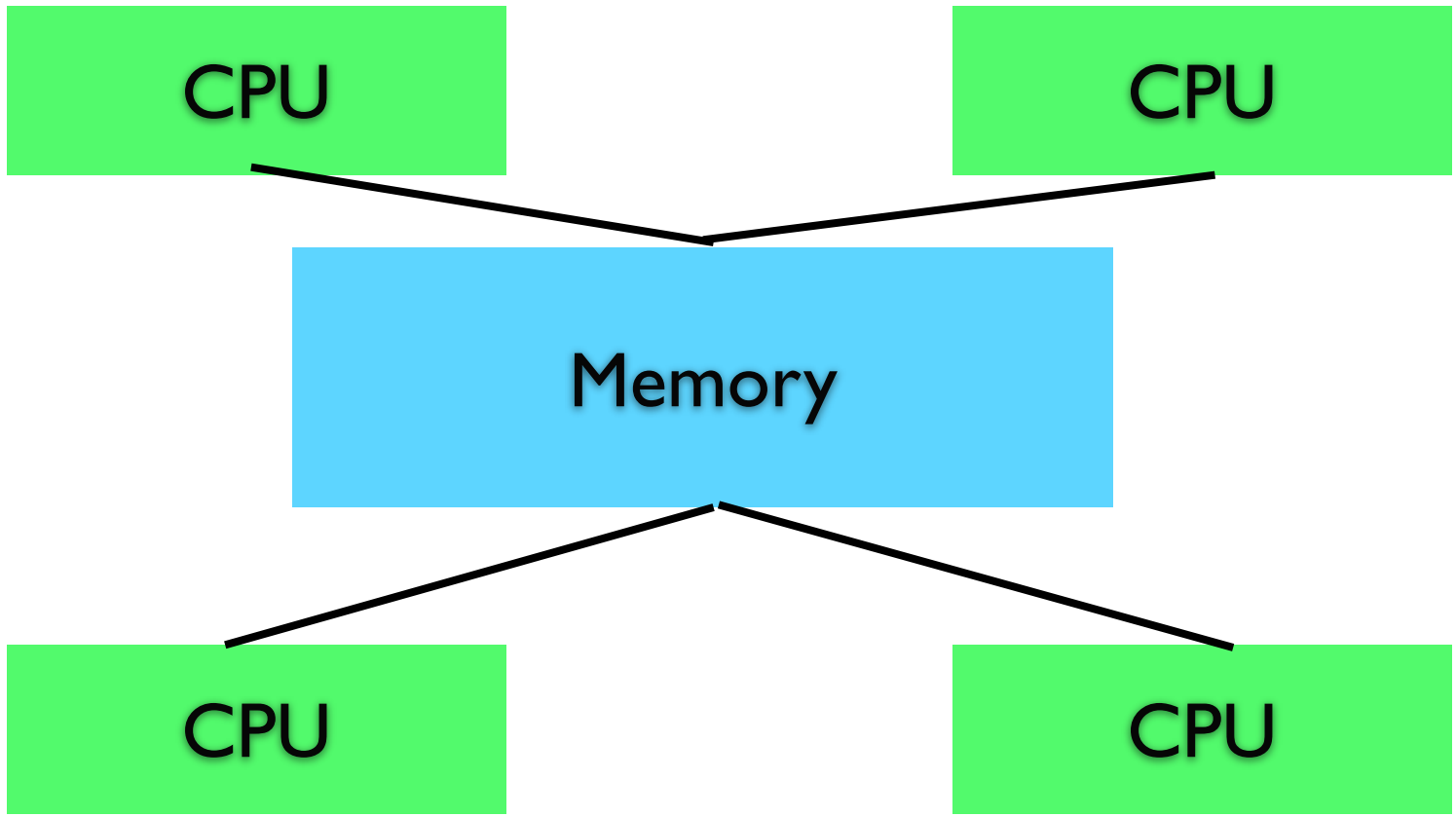
Shared memory architectures



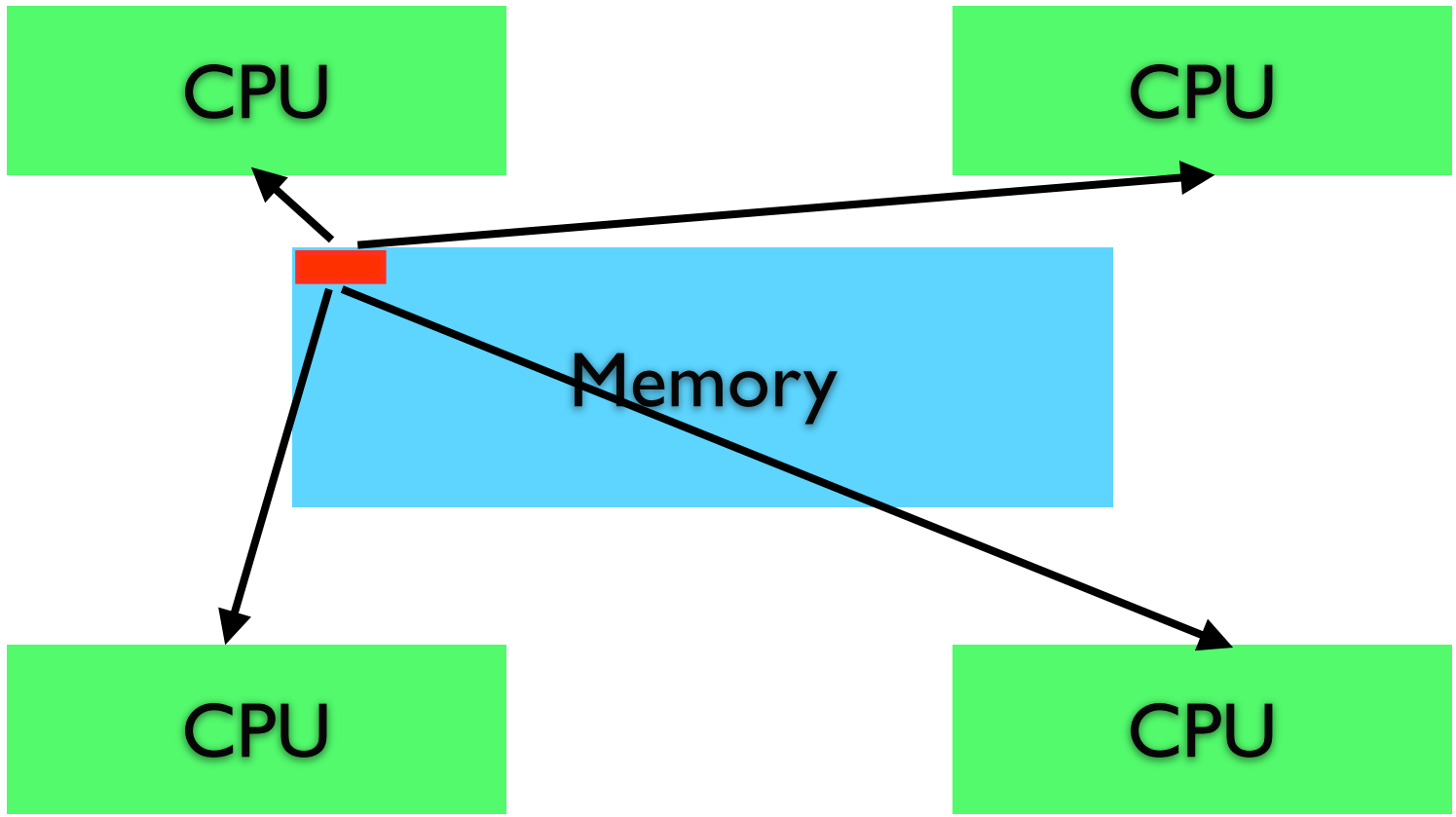
All processors identical



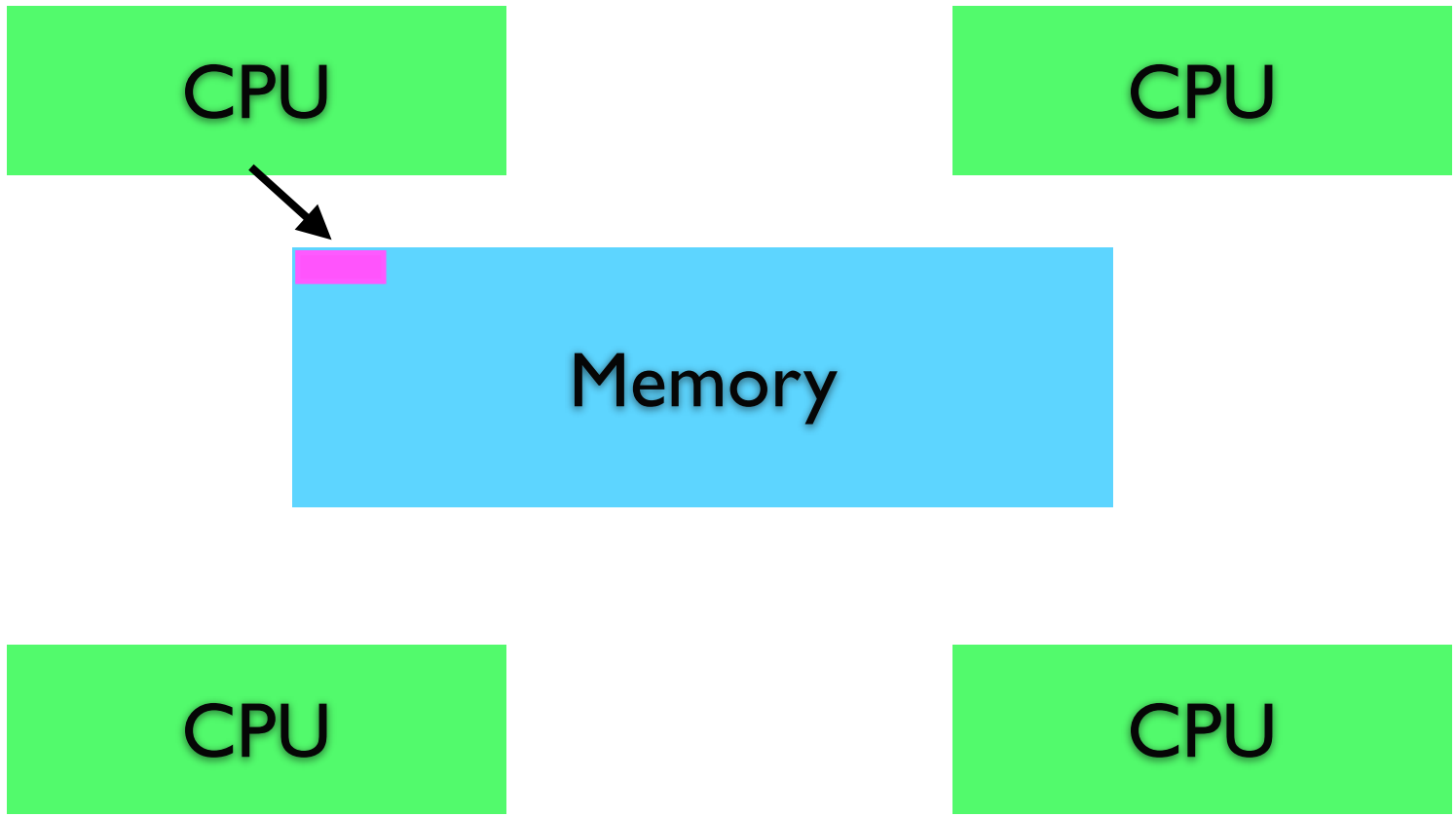
Equal access time to all memory



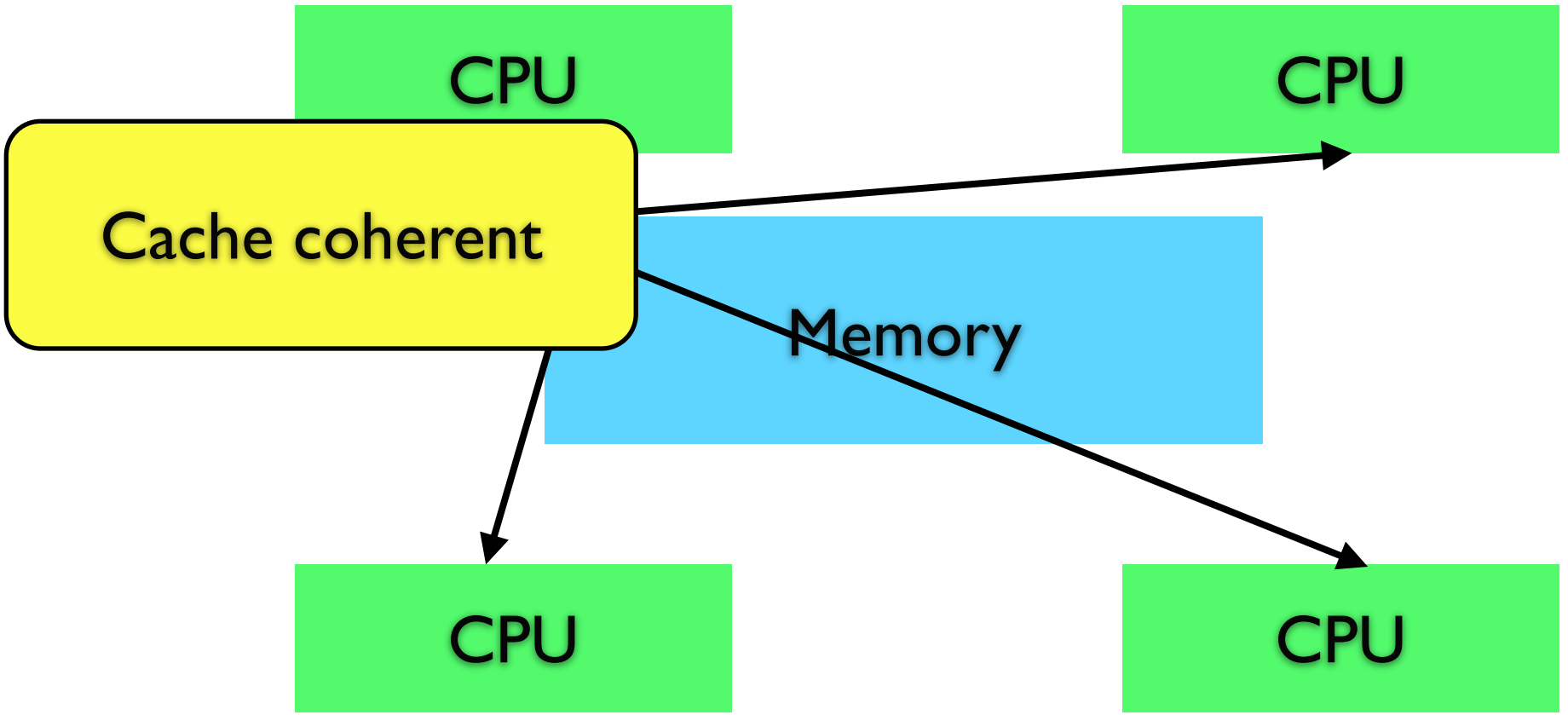
All processors can simultaneously access



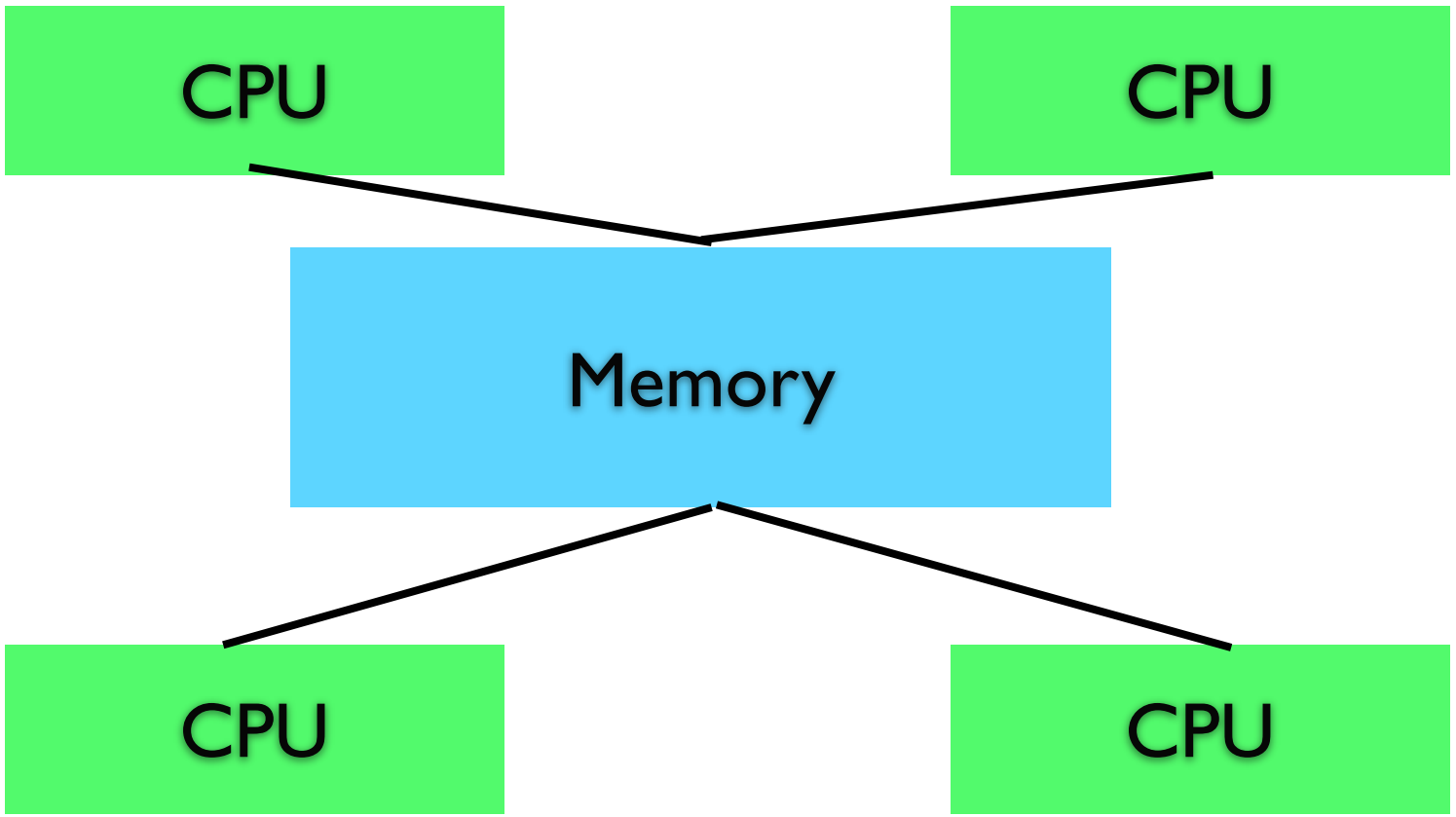
Changes by one processor is seen by all



Changes by one processor is seen by all



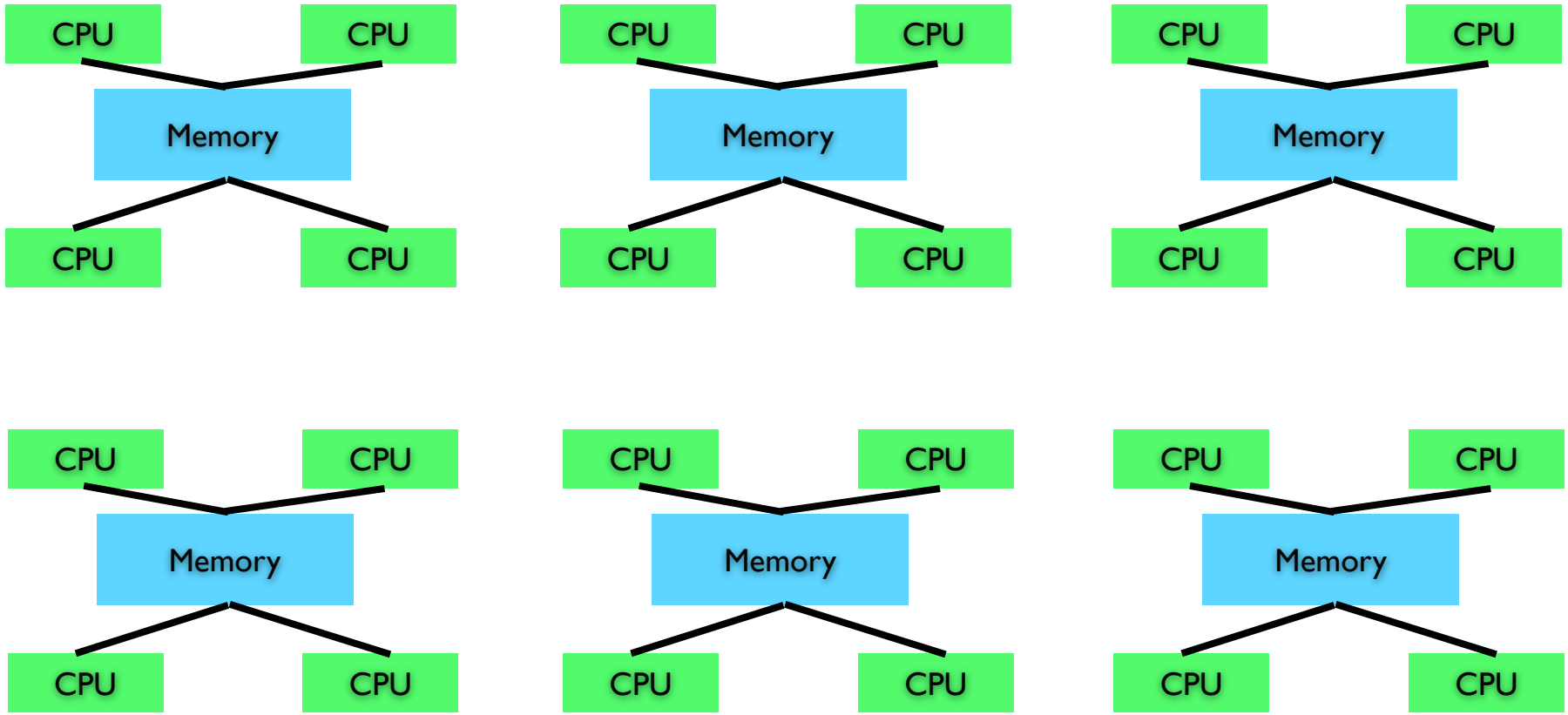
Uniform Memory Access (UMA)



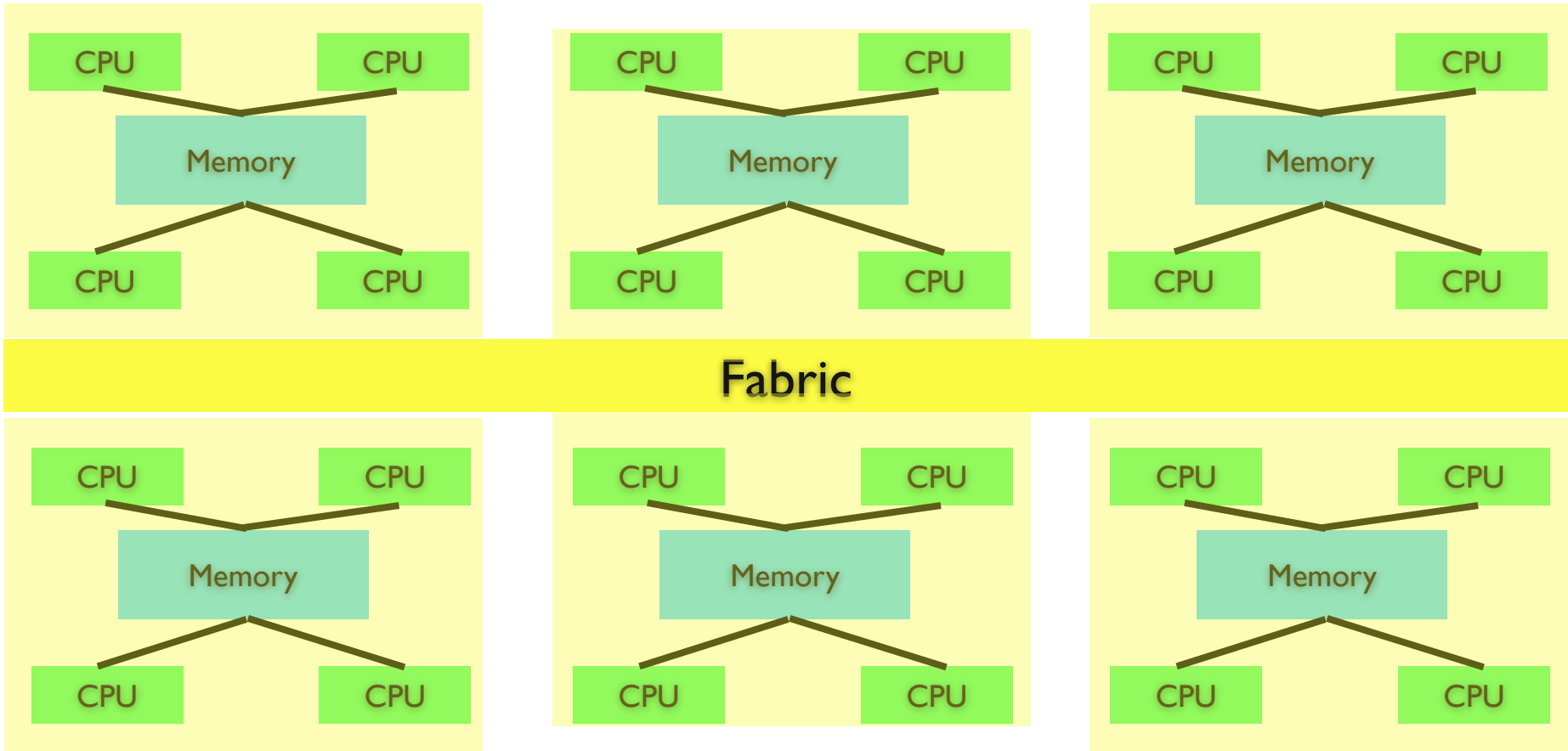
SMP Advantages

- Fast access to all memory locations
- Easy to program

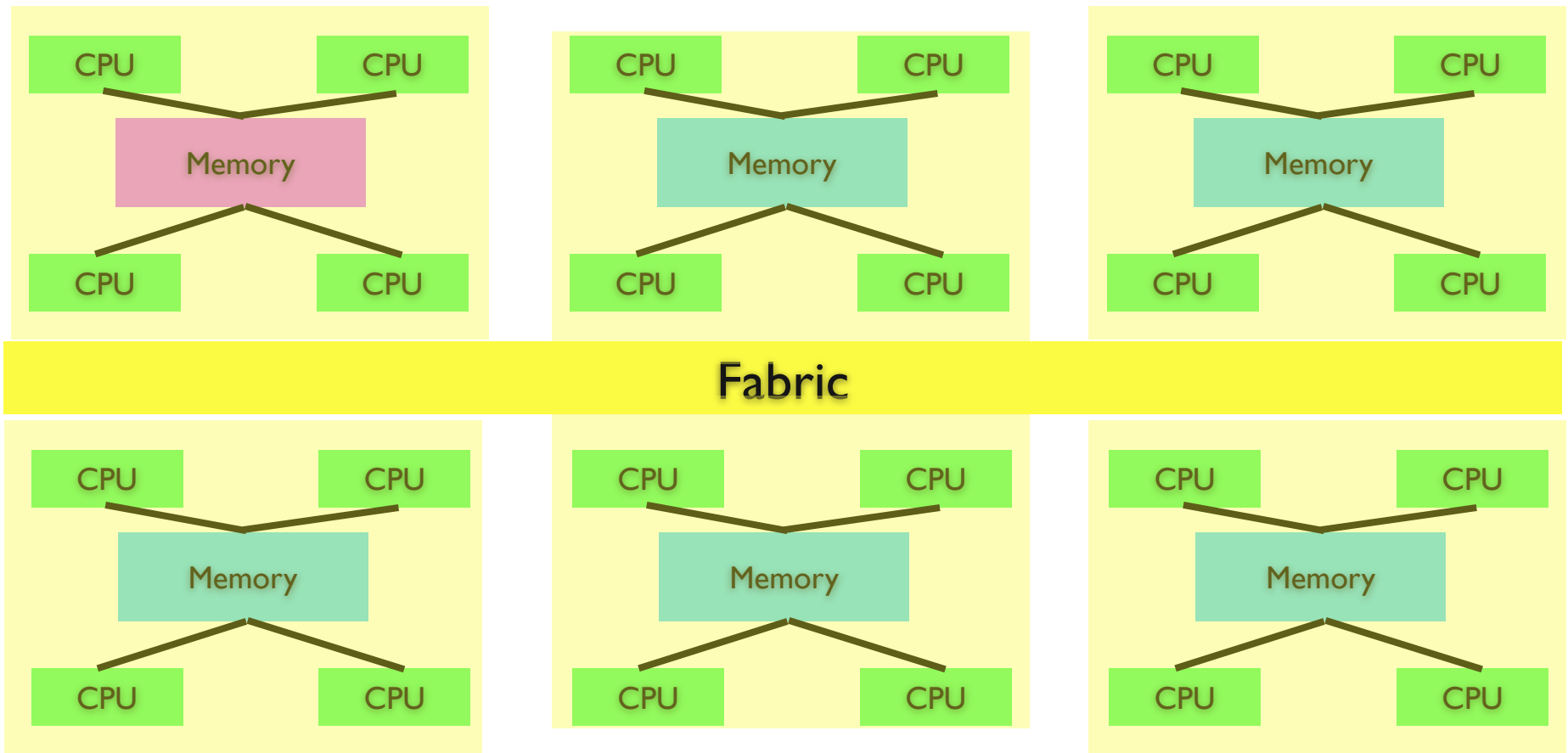
Non-uniform memory access



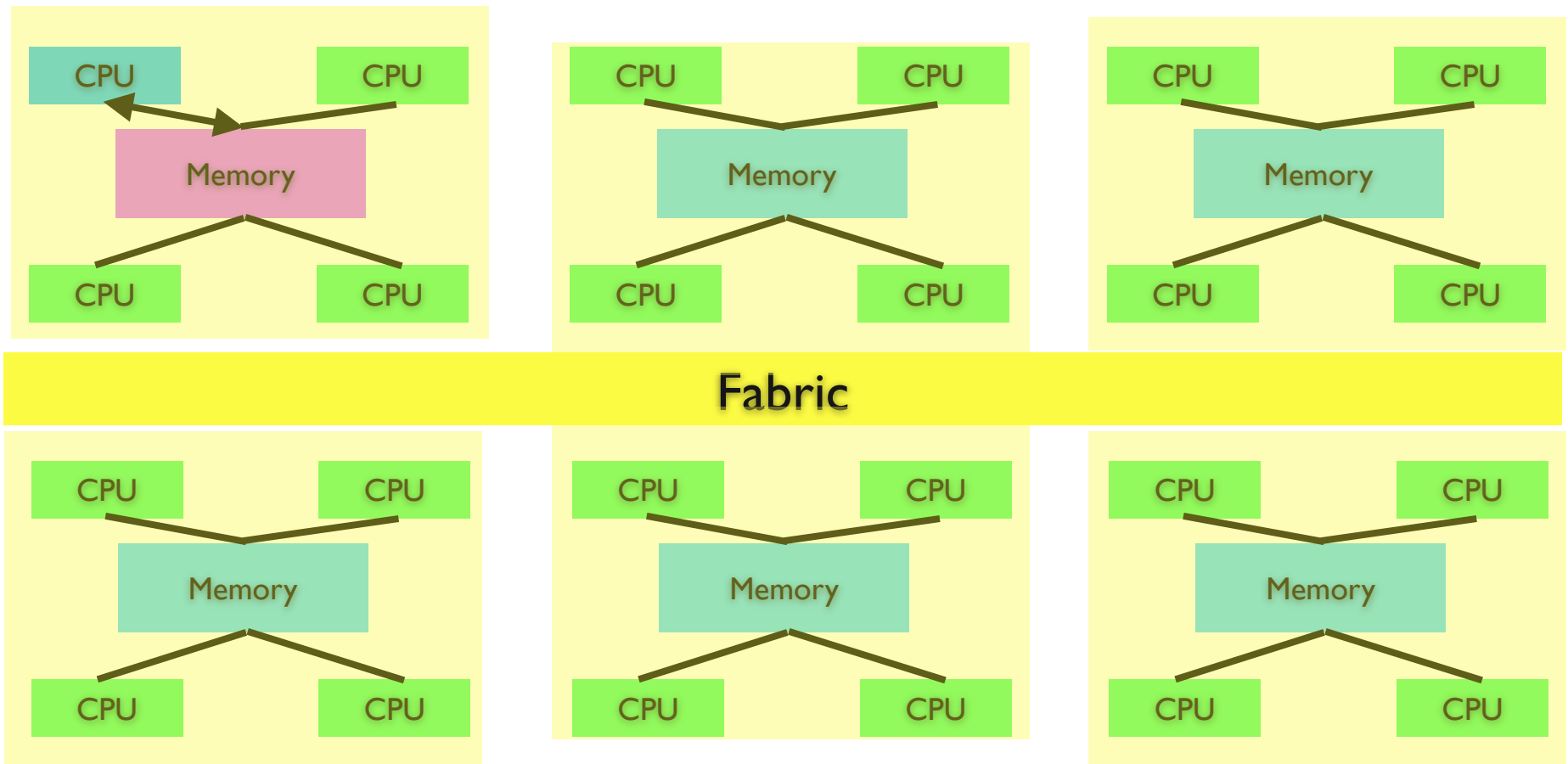
Non-uniform memory access (NUMA)



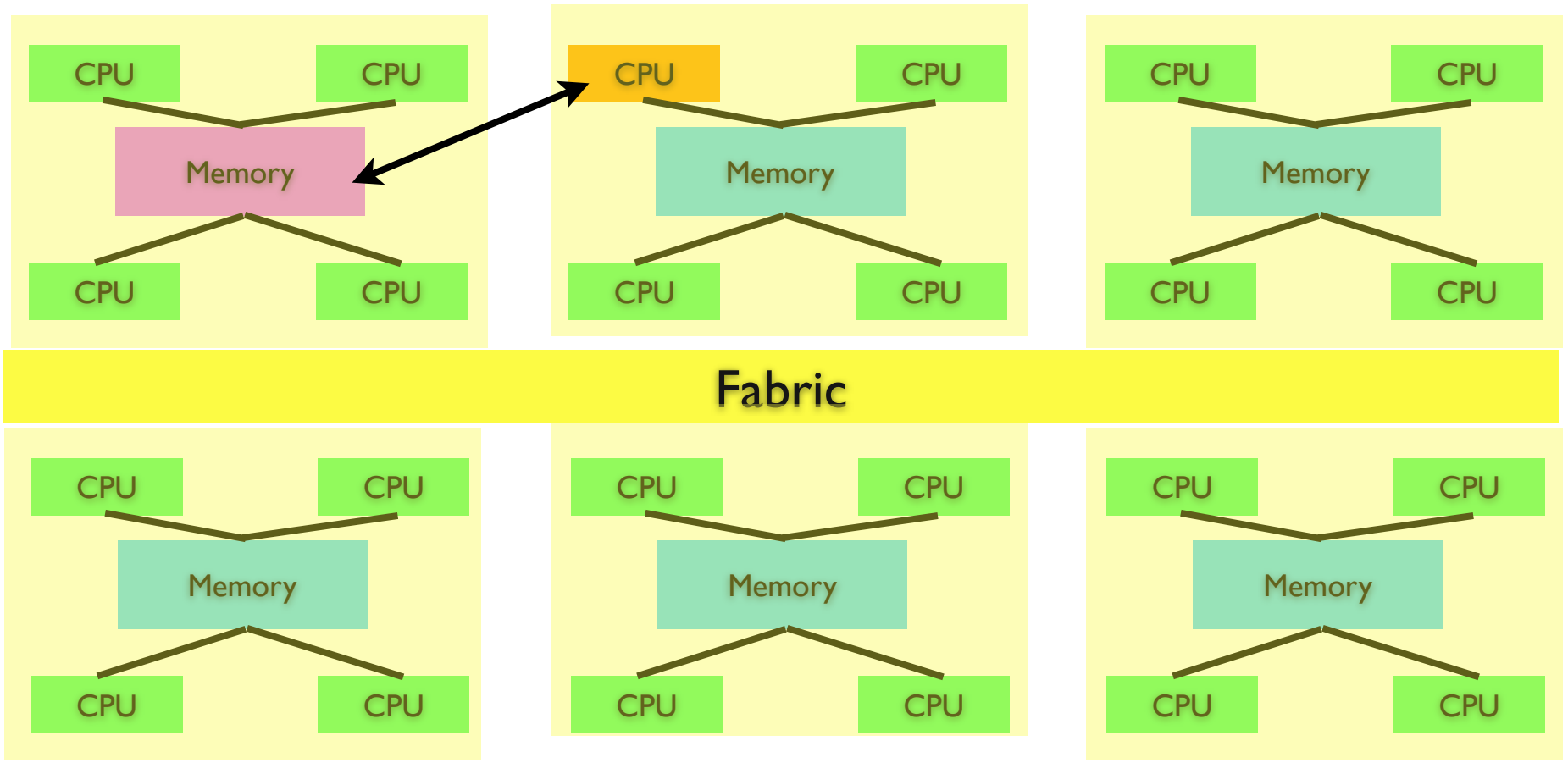
Local memory



Local memory



Distant memory



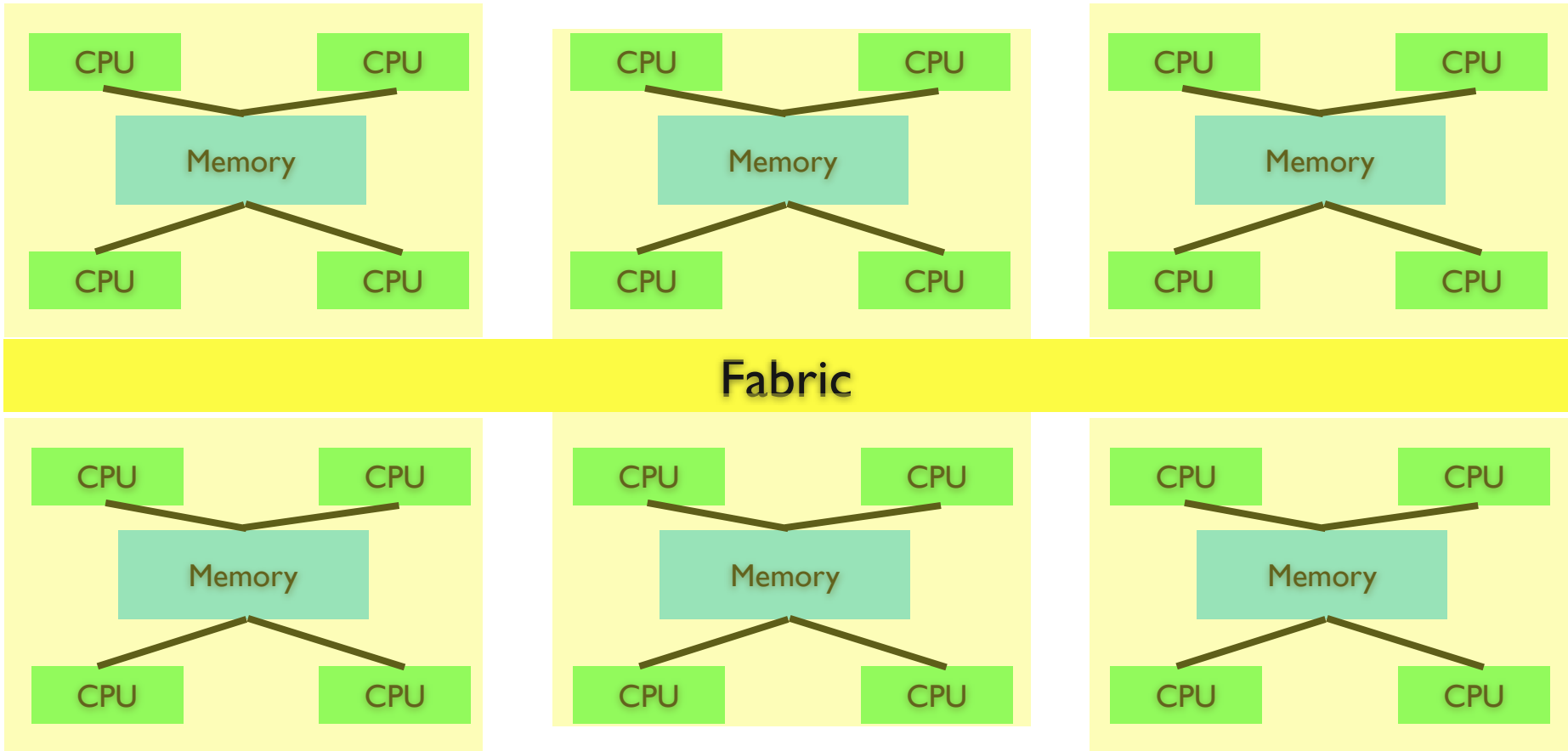
SMP Disadvantages

- If written poorly a code on NUMA machine will slow down with increasing processors quickly
- Limited scalability at a reasonable cost

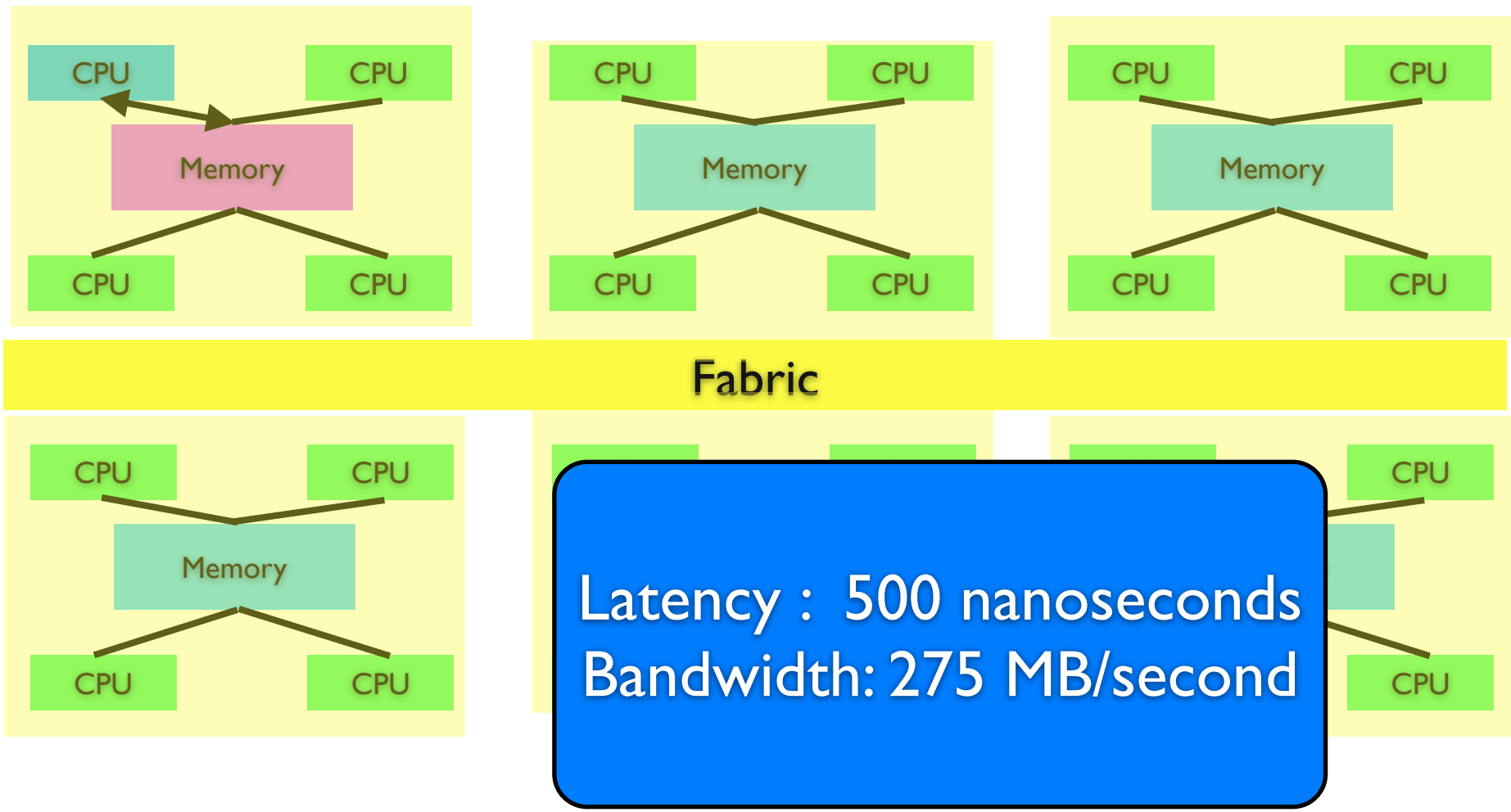
Terminology

- Latency - the time it takes to receive the first byte
- Bandwidth- how many bytes can be transferred in a given amount of time

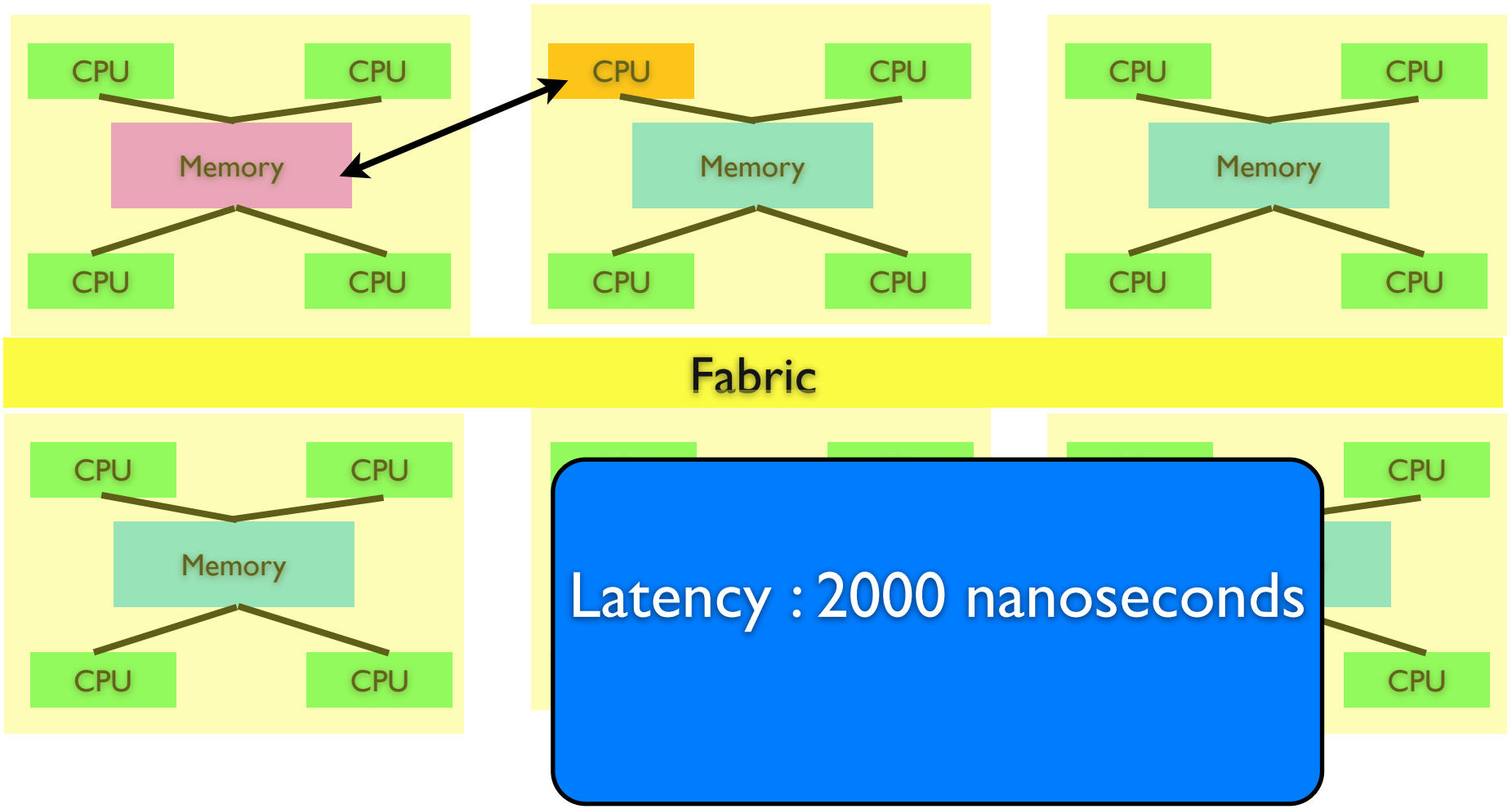
Sun Fire 6900



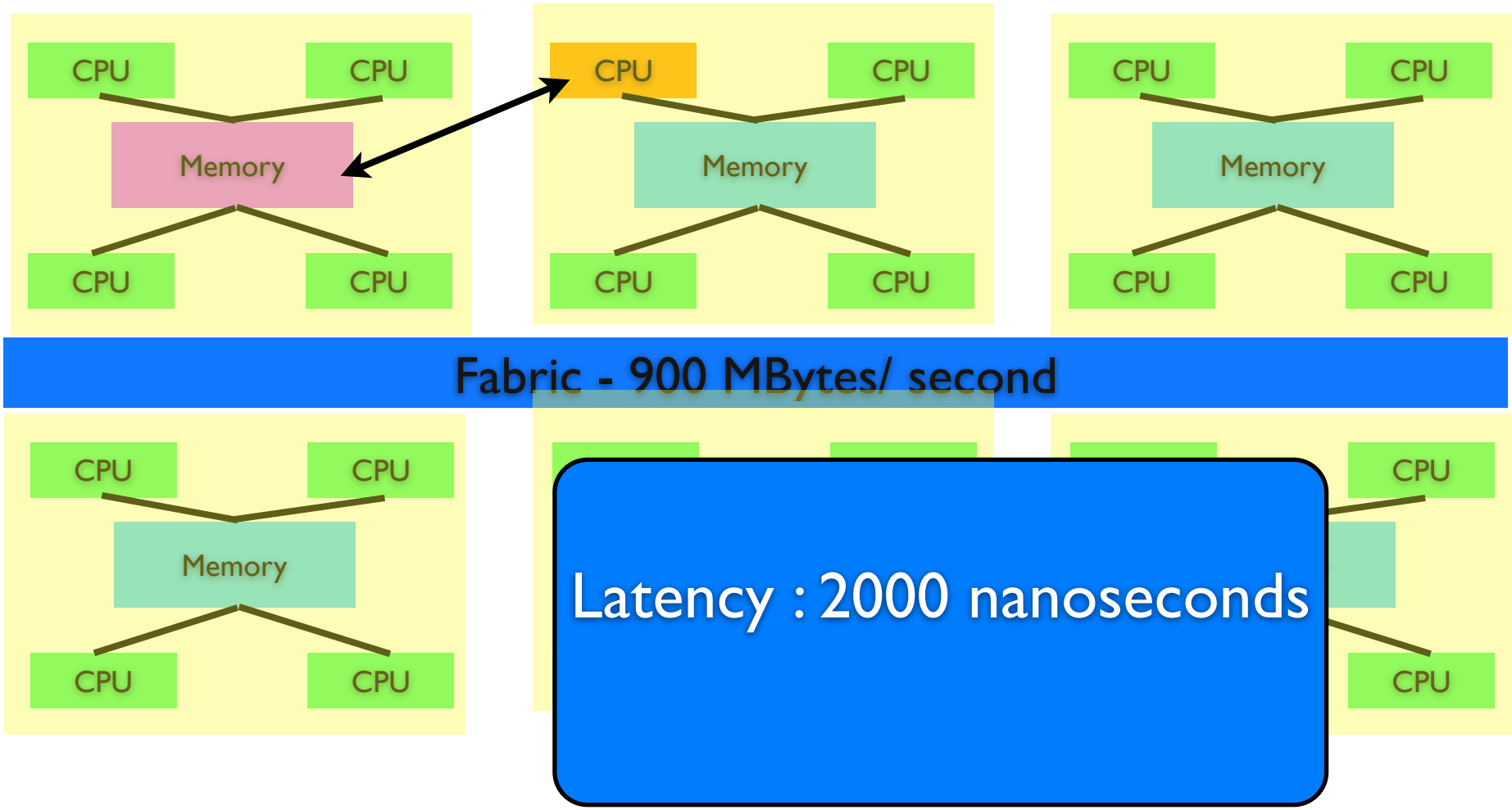
Sun Fire: Local memory



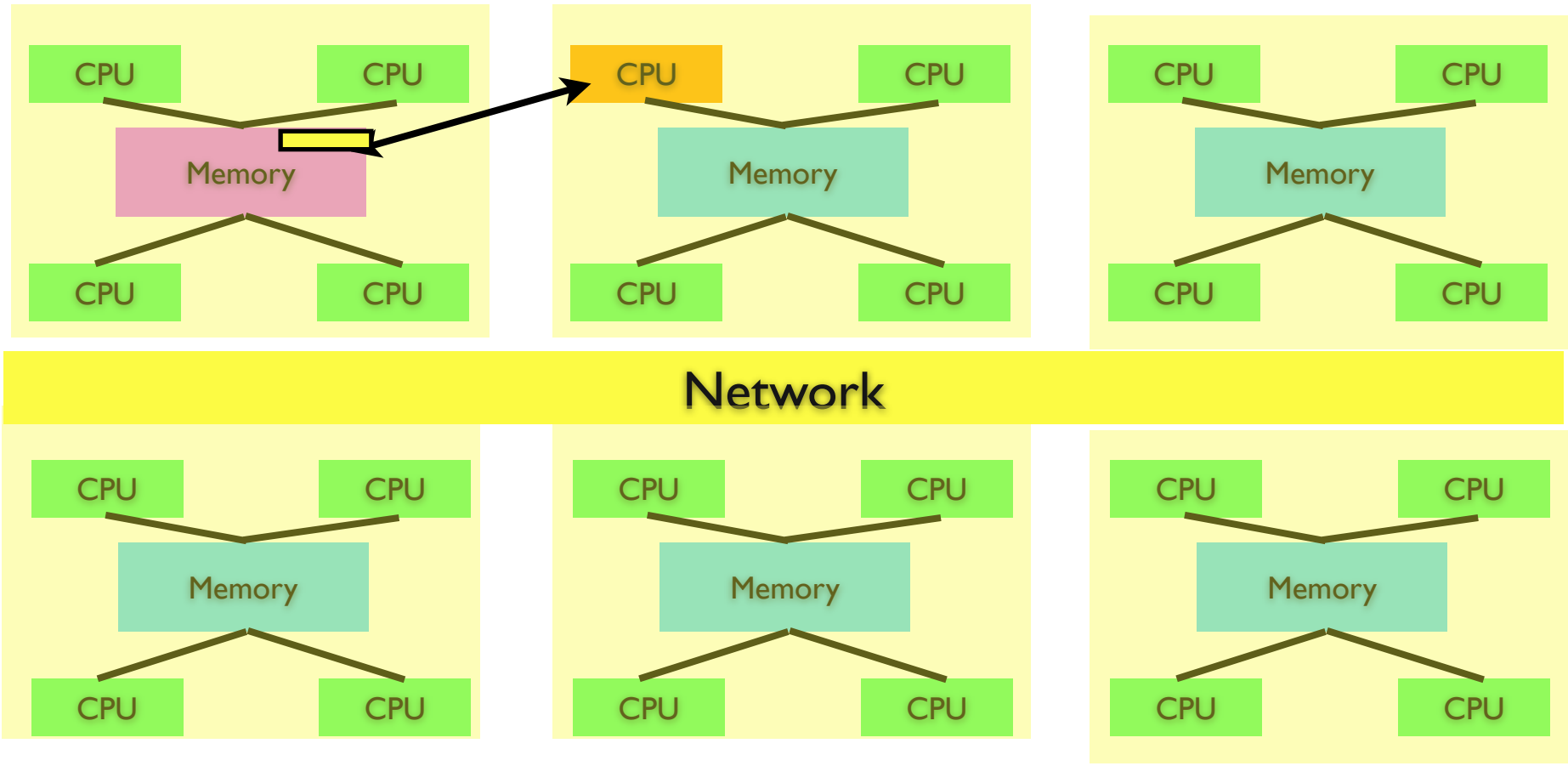
Sun Fire: Distant memory



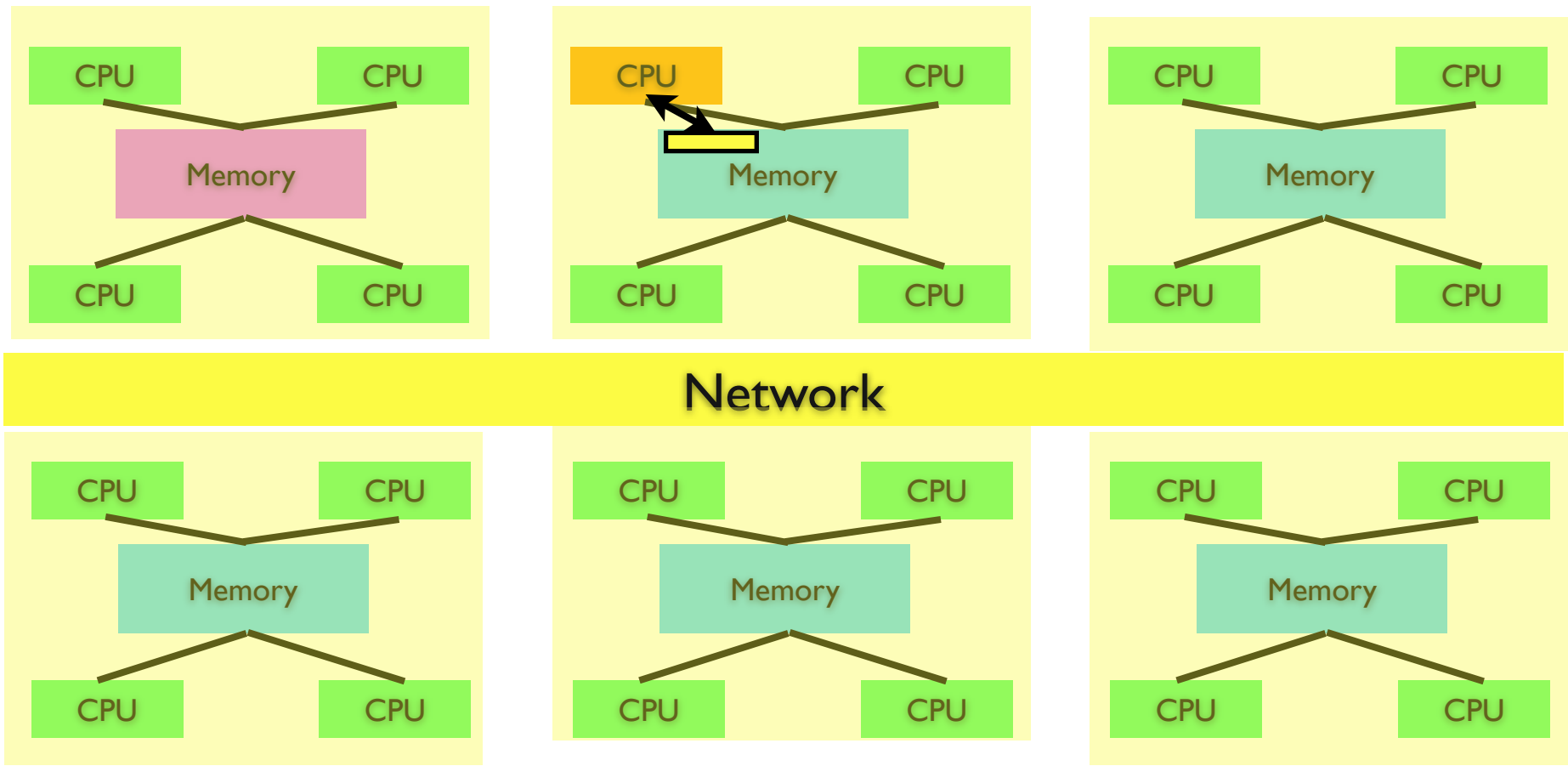
Sun Fire: Distant memory



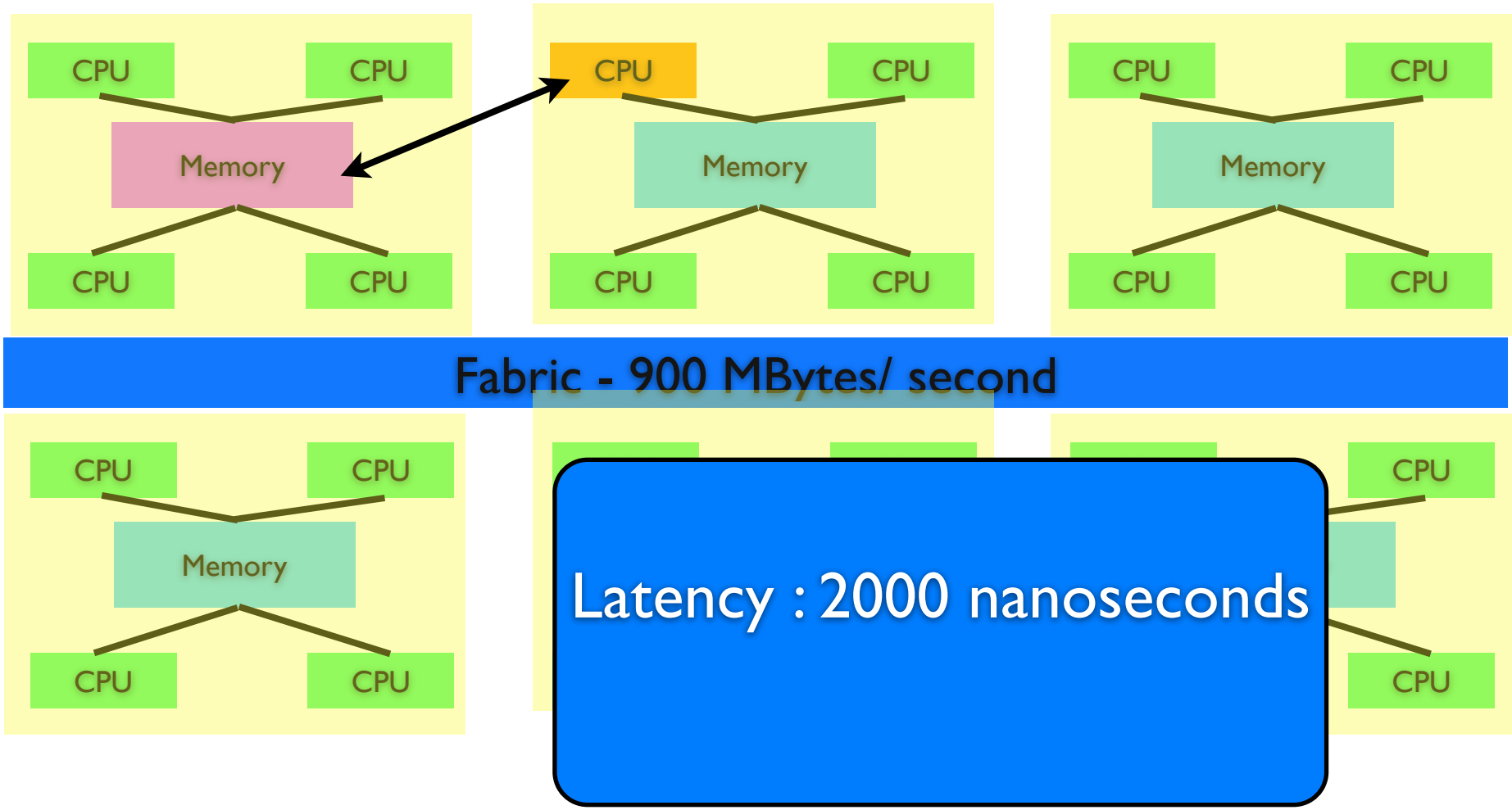
Sun Fire: Memory migration



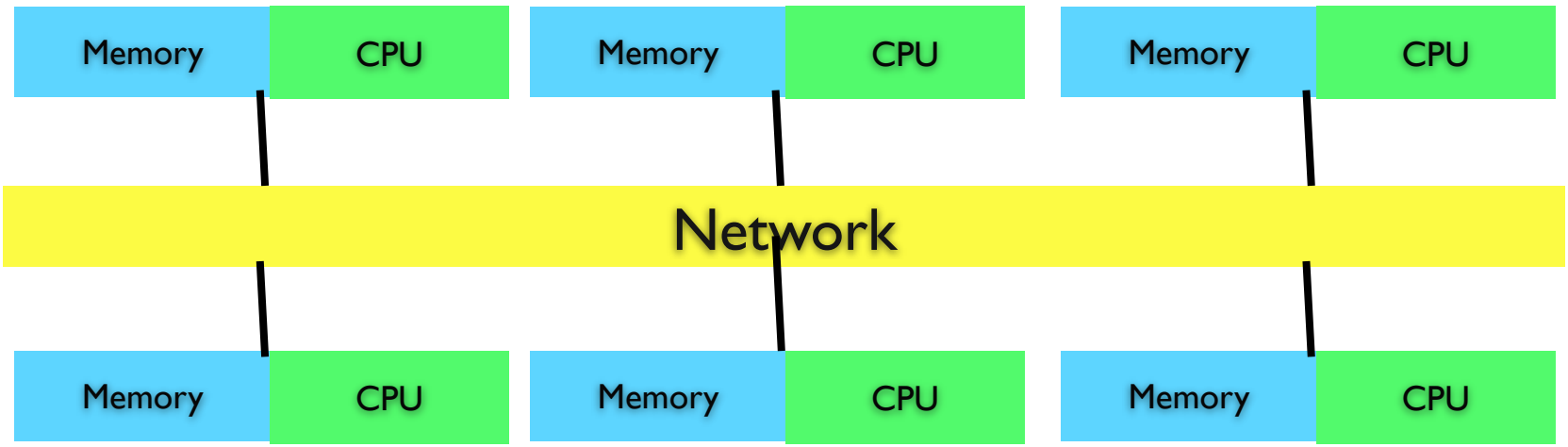
Sun Fire: Memory migration



Sun Fire: Increased nodes, increase fabric complexity



Distributed memory



Architectures

Architecture	UMA	NUMA	Distributed
Example	SMP	Sun 6900	Beowulf cluster
Communication	MPI, threads, Openmp	MPI, threads, Openmp	MPI
Scalability	10s of processors	100s of processors	1000s of processors
Drawbacks	Memory bandwidth	Non-uniformity	programming challenge

Distributed memory: Advantages

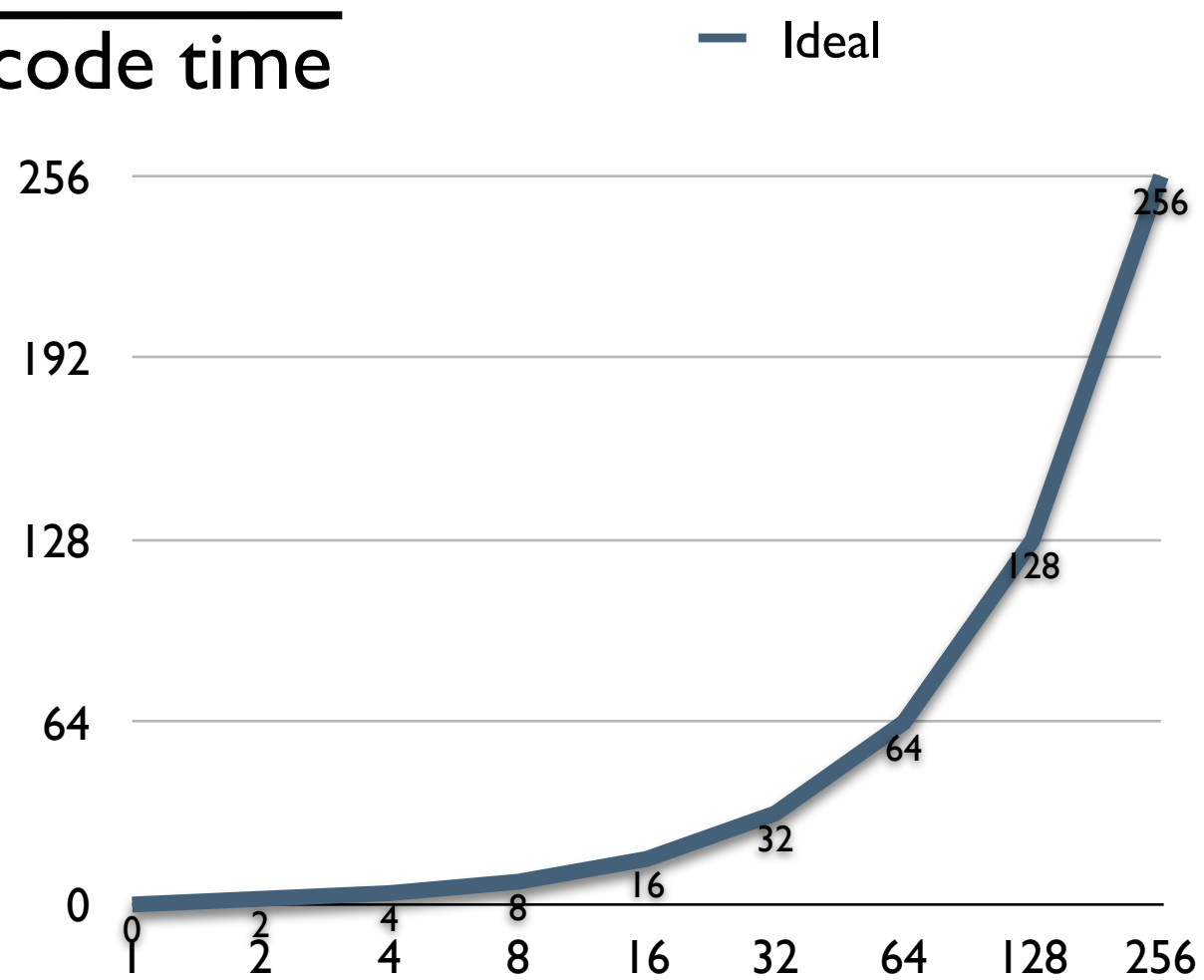
- For certain problems (embarrassingly parallel) scales very well
- Cheap (computers are commodity hardware)
- Memory scales with the number of processors

Measuring speedup

$$\text{speedup} = \frac{\text{serial code time}}{\text{parallel code time}}$$

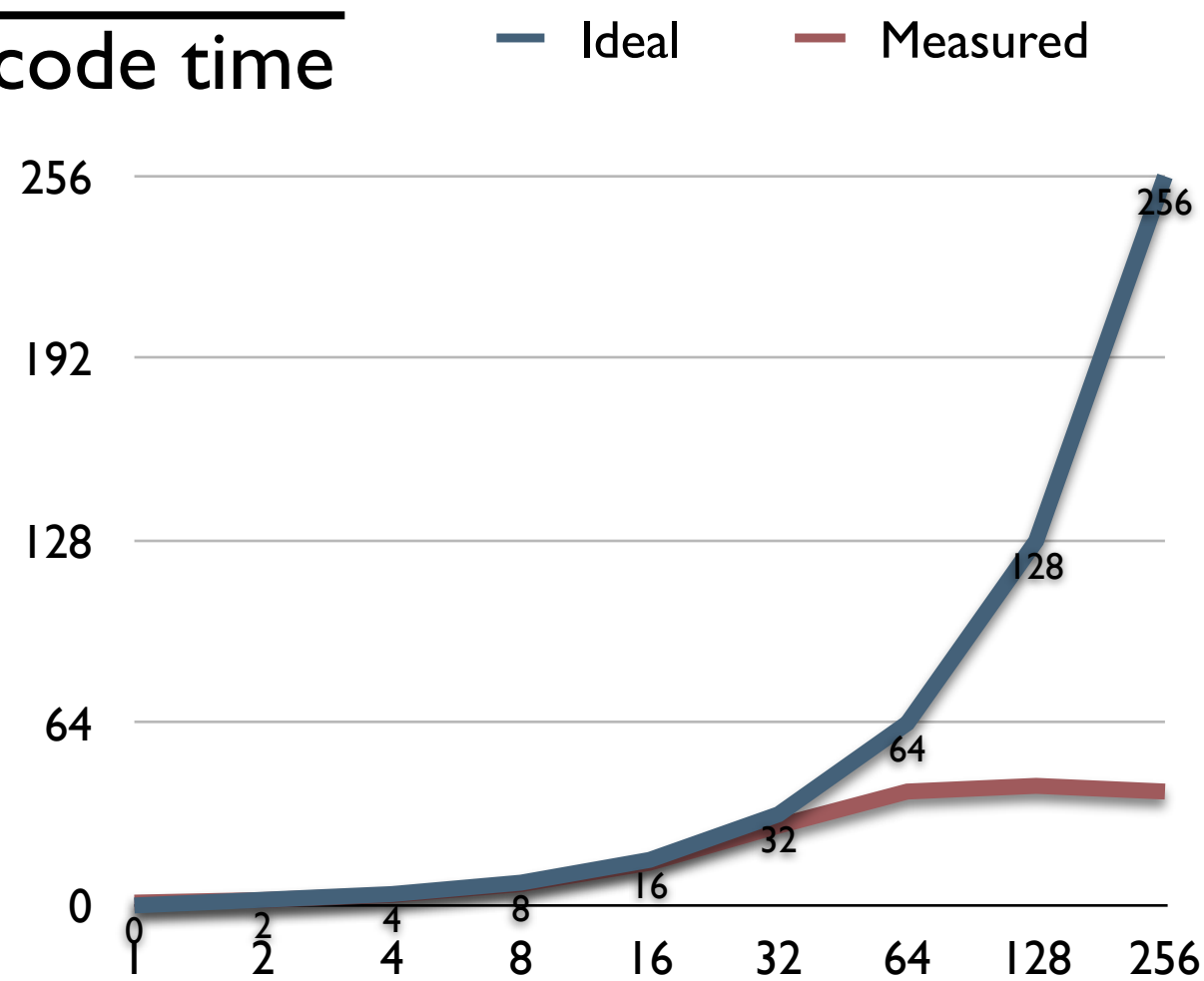
Perfect speedup

$$\text{speedup} = \frac{\text{serial code time}}{\text{parallel code time}}$$



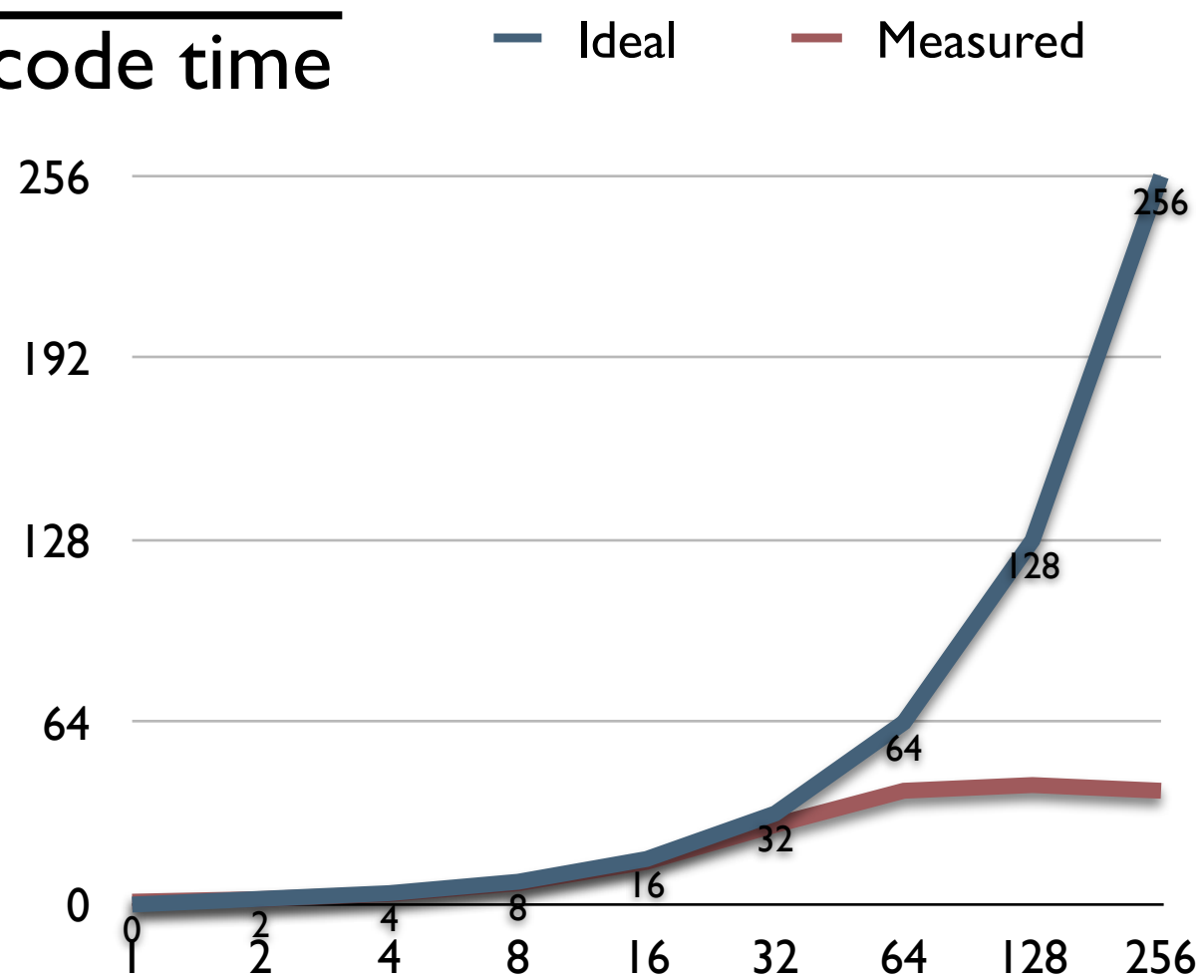
Typical pattern

$$\text{speedup} = \frac{\text{serial code time}}{\text{parallel code time}}$$



Typical pattern

$$\text{speedup} = \frac{\text{serial code time}}{\text{parallel code time}}$$

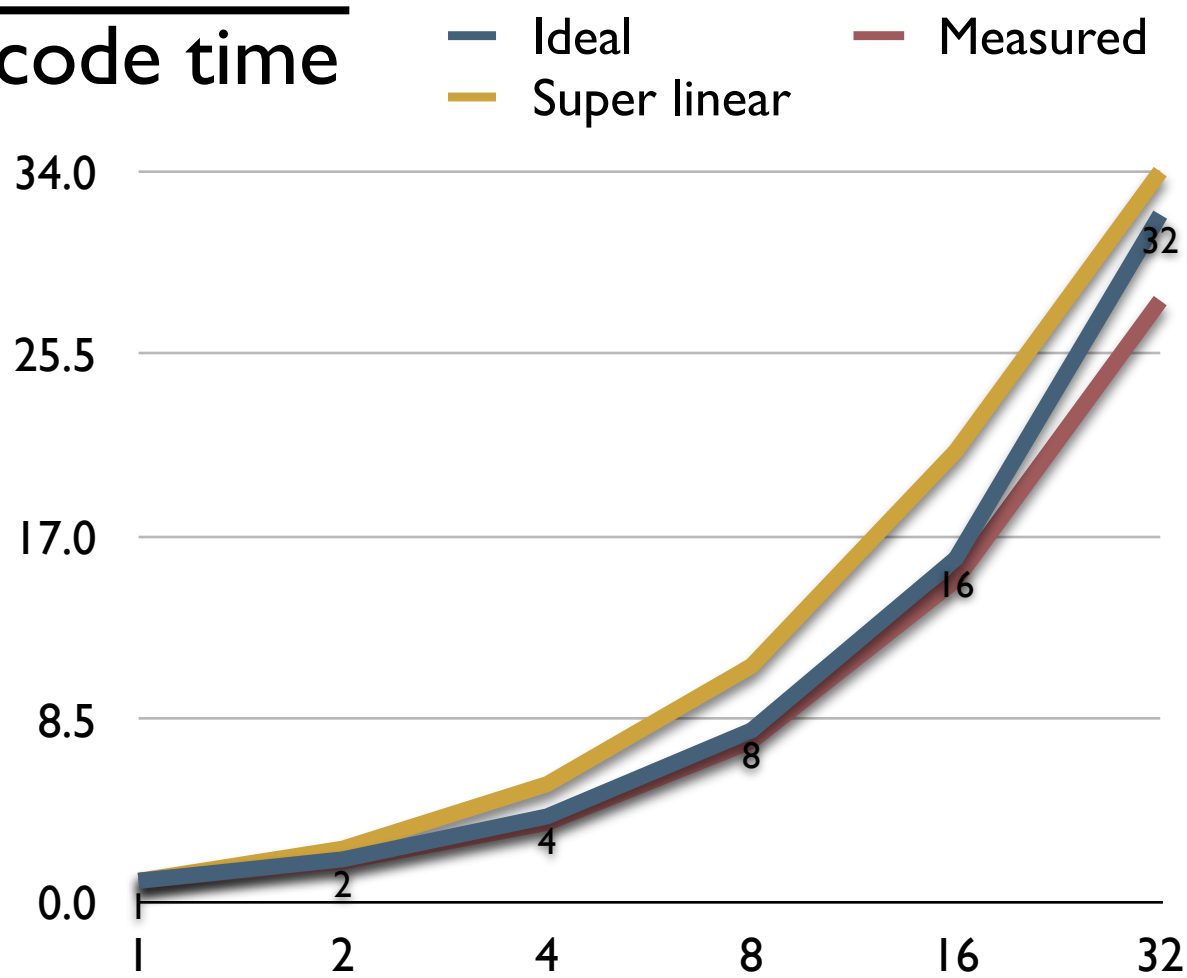


Note the maximum

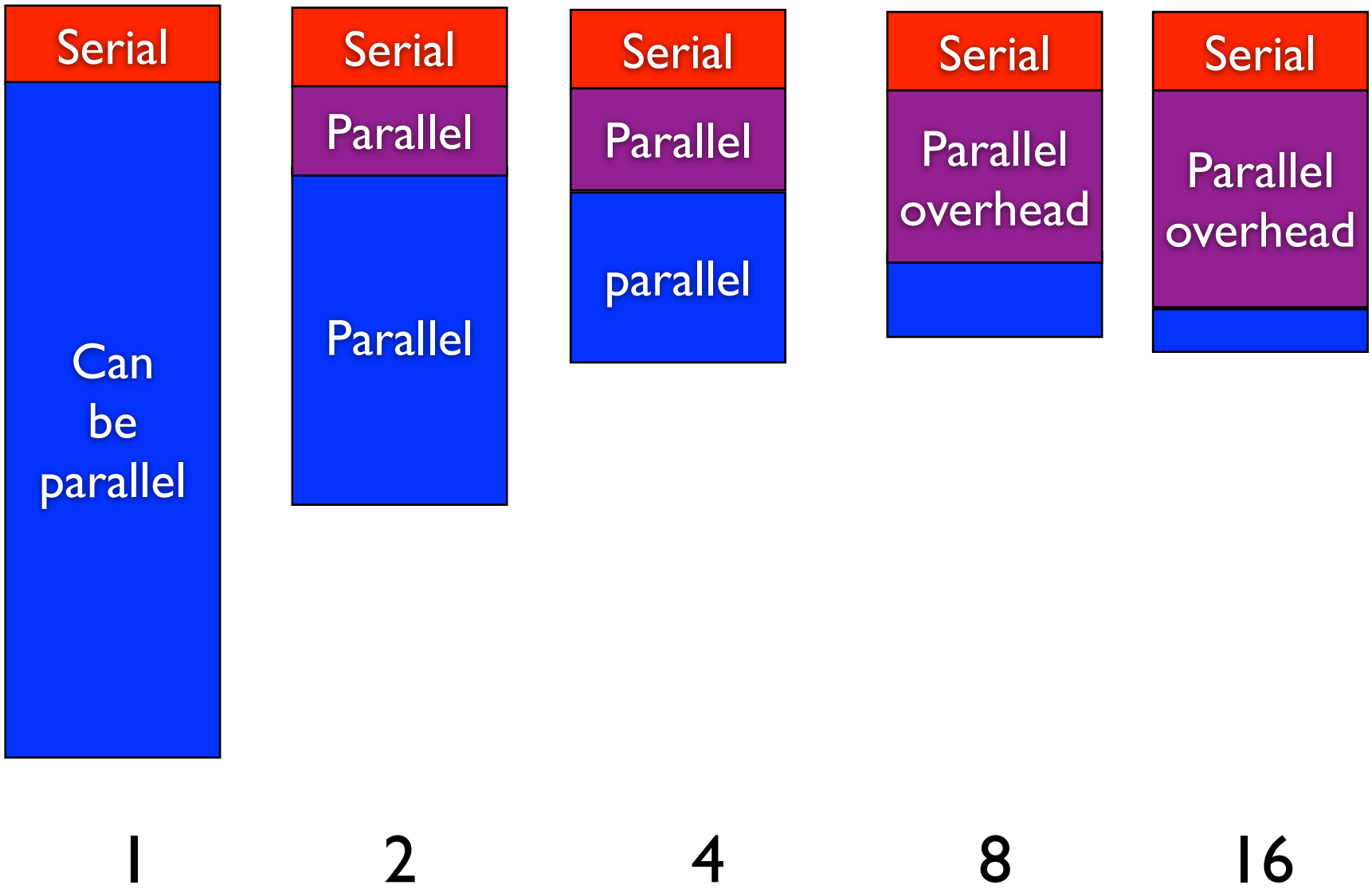
Super linear

$$\text{speedup} = \frac{\text{serial code time}}{\text{parallel code time}}$$

Example:
Better cache
behavior



Clock time

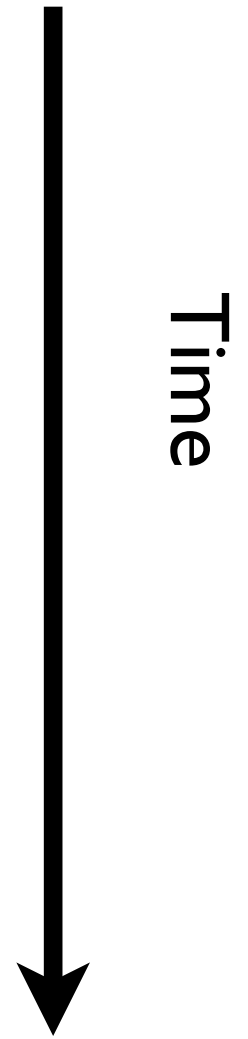

Serial
Can be parallel
Serial
Parallel
Parallel
Serial
Parallel
parallel
Serial
Parallel overhead
Serial
Parallel overhead
1
2
4
8
16

Programming models

- Threads
- Message passing
- Data parallel model
- Collection of tasks

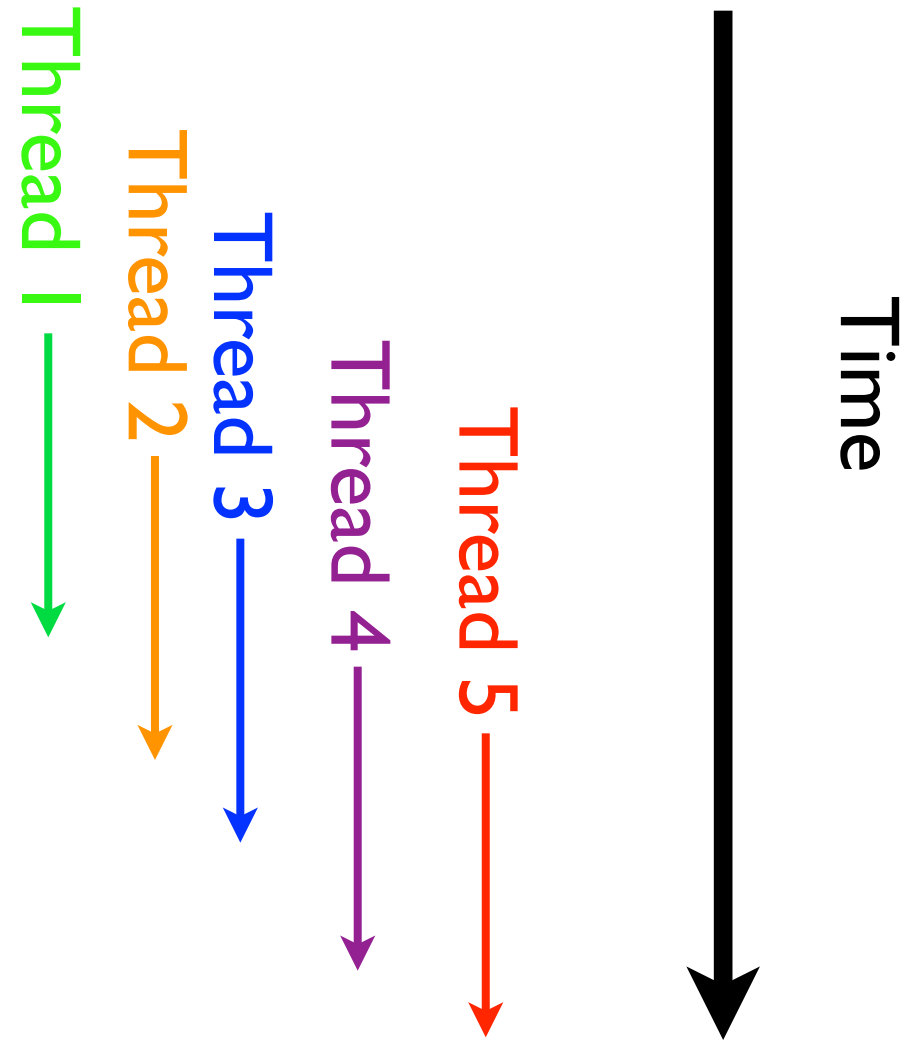
Threads model

```
program alpha
.
.
call sub1()
call sub2()
call sub3()
call sub4()
call sub5()
.
.
end program
```



Threads model

```
program alpha
.
.
call sub1()
call sub2()
call sub3()
call sub4()
call sub5()
.
.
end program
```



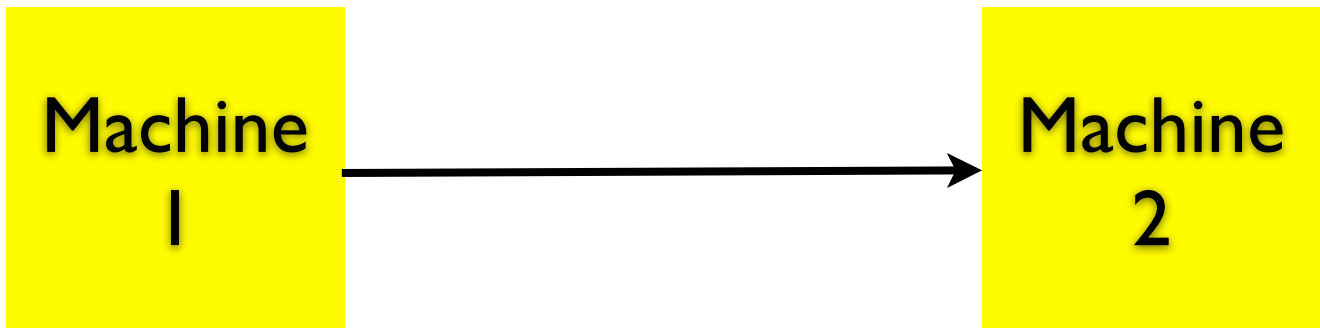
Thread model implementation

- POSIX Thread library
  - Always available
  - Directly only in C
  - Hard, lots of ways to mess up
- OpenMP
  - Compiler directives
  - Fortran, C, C++
  - Easier, less ways to mess up

Message passing

```
if(impi==0) {
send(buf,nbuf,1)
.
}
if(impi==1){
recv(buf,nbuf,1)
,
}
```

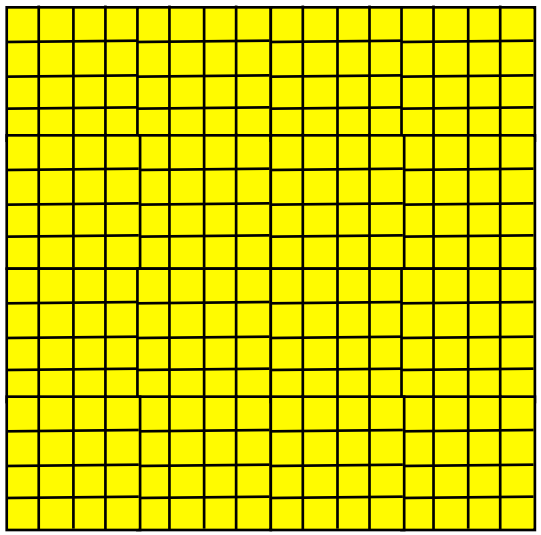
Machines explicitly pass data



Message passing

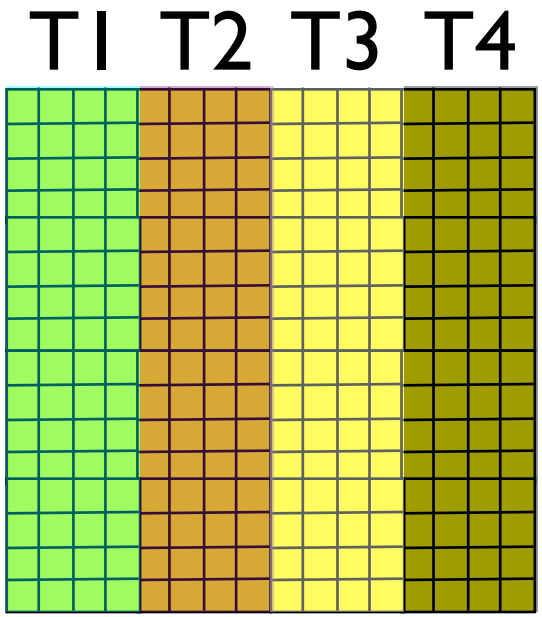
- Parallel Virtual Machine (PVM)
  - Dying
- Message Passing Interface (MPI)
  - MPICH, OpenMap, Lam/MPI, etc.
  - Approximately same level of complexity as POSIX threads

Data parallel model



```
program alpha
real :: alpha(:,:)
```

Data parallel model




```
program alpha
real :: alpha(:,:)
!HPF$ Distribute alpha(BLOCK,*)
```

Data parallel model

- Implementation
  - CM Fortran
  - High Performance Fortran
  - Unified Parallel C
  - Co-Array Fortran
- Automatic mapping to SMP and distributed memory
- Easy to program, not much used, maybe coming back

Hybrid models

- Combine two or more mechanisms
  - MPI and Posix, MPI and OpenMP
- Model of the future on conventional hardware
  - Many cores/processors on each machine

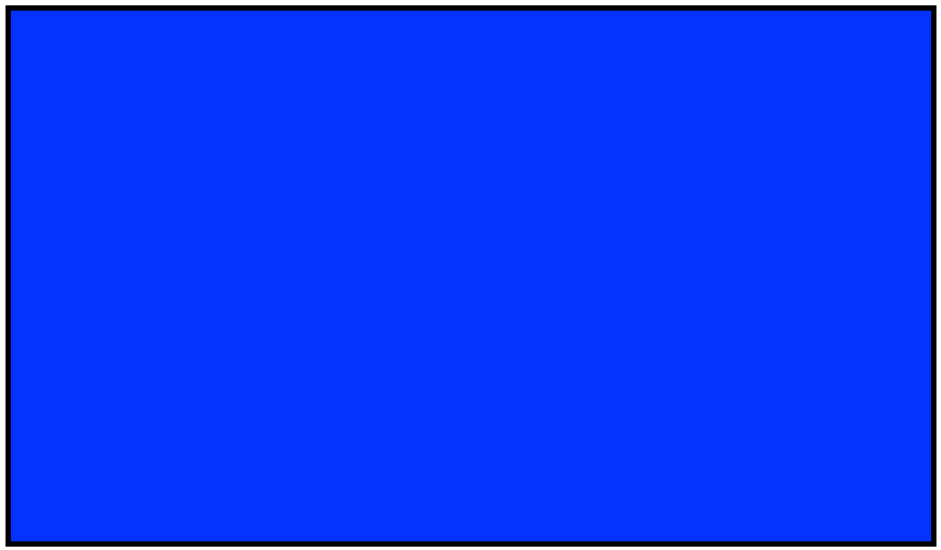
Automatic parallelization

- Many compilers claim they can automatically parallellize your code
- Very simple loops (think vector operations) will work
- Anything more complex can
  - Run slower
  - Produce errors

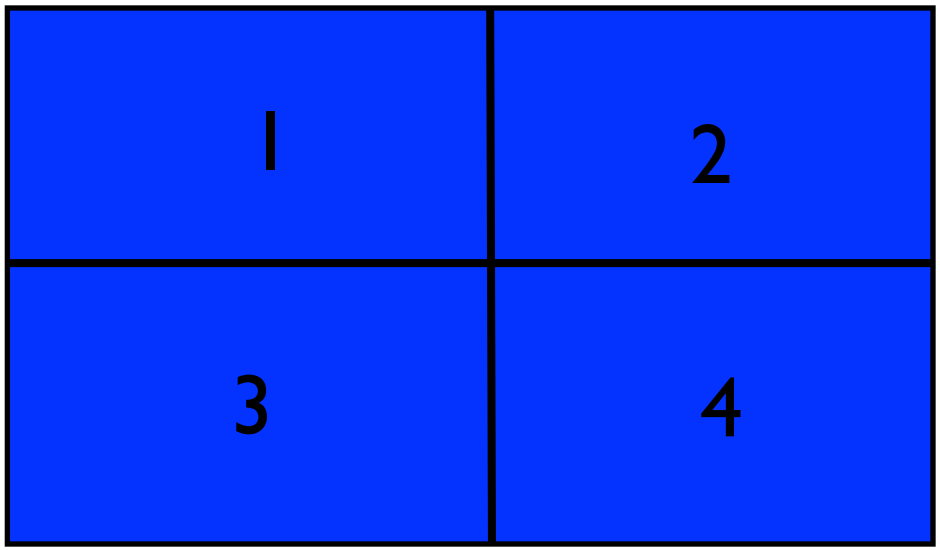
Partitioning

- Domain decomposition
- Functional decomposition

Domain decomposition

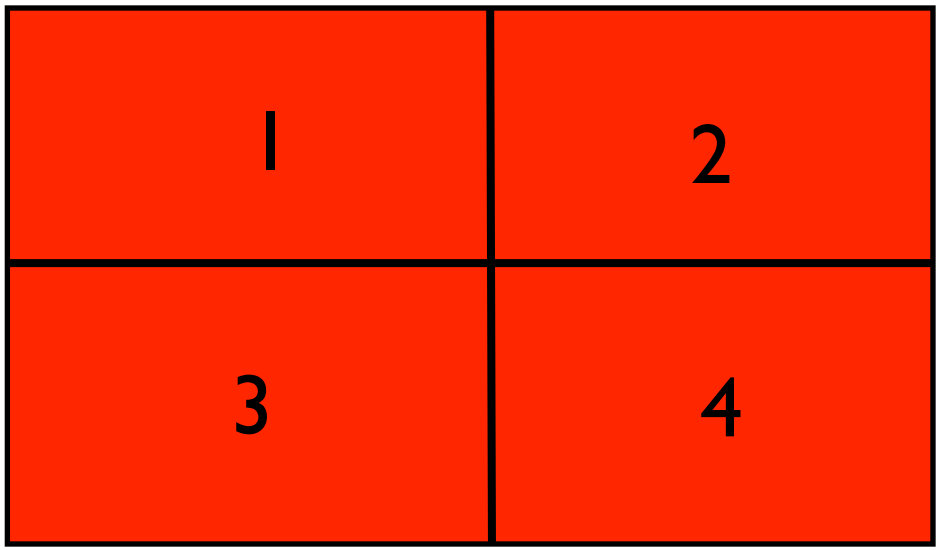


Domain decomposition



Different processors are assigned different portions of the dataset to work on

Functional decomposition

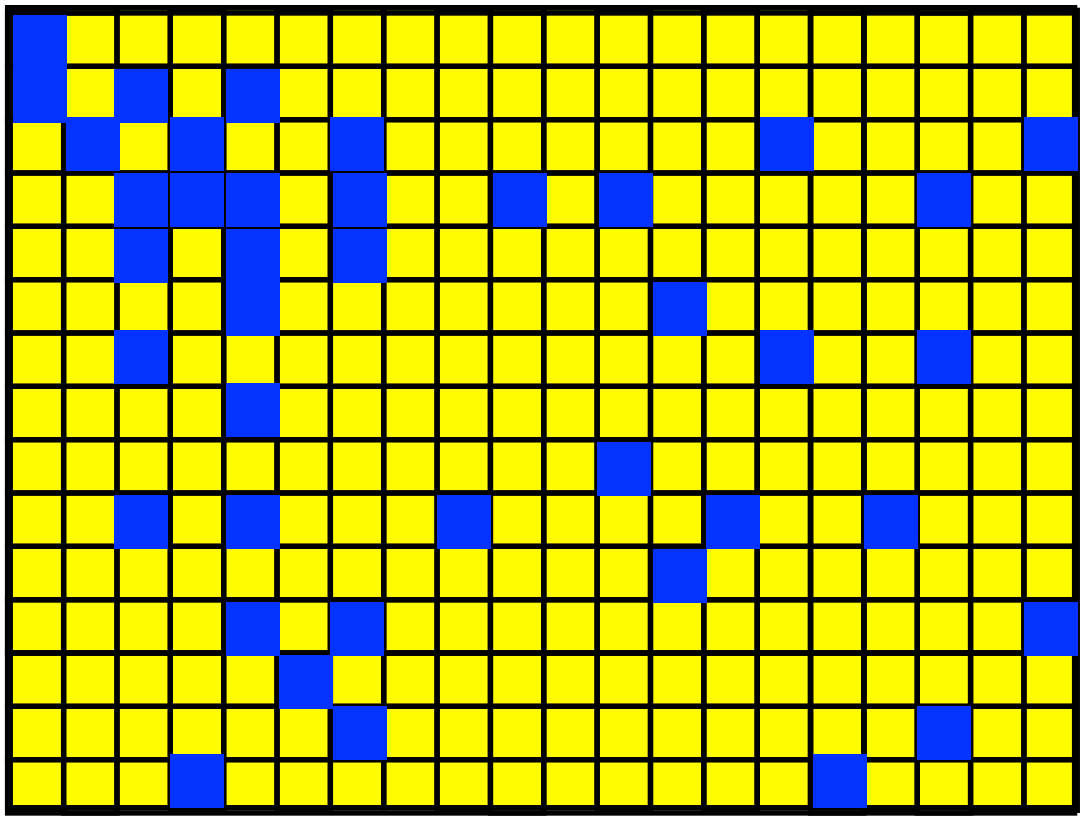


Different processors are assigned different tasks to work on

How many processors?

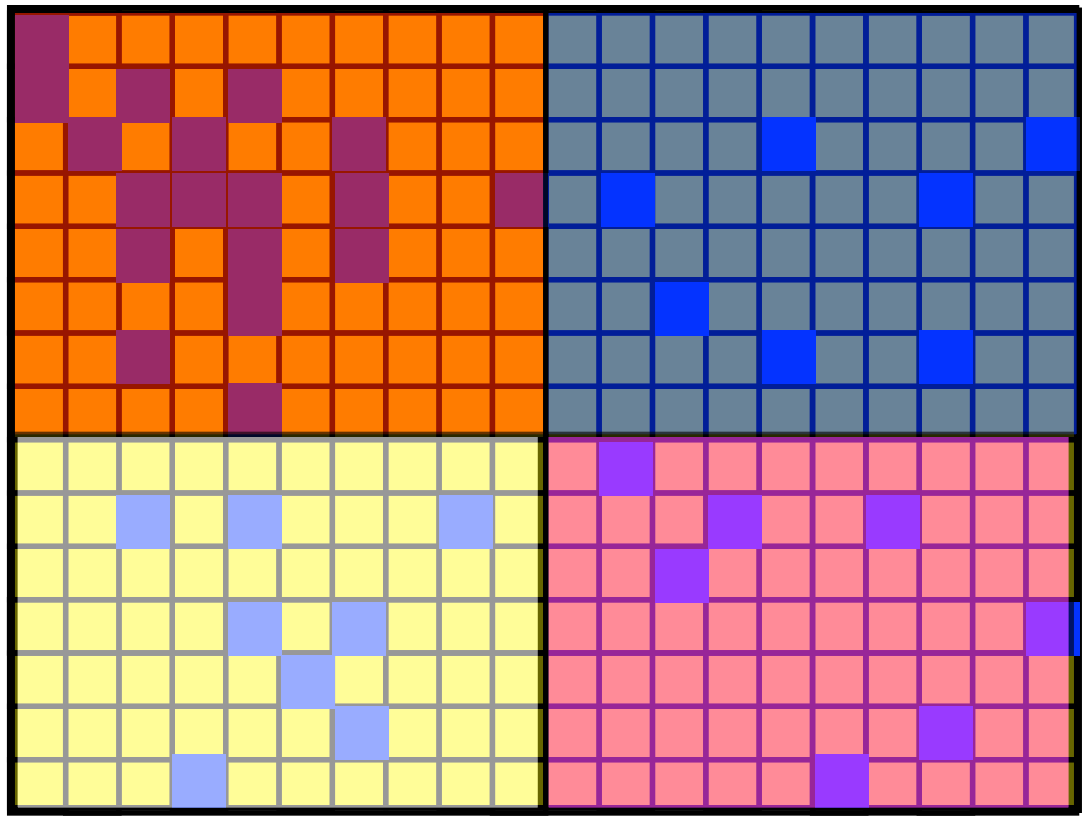
- Functional decomposition is generally more limited (how many different task do you have?)
- Both are dependent on the amount of communication/synchronization needed

Sparse matrix multiplication



```
for i in range(npts):
        dat[iloc[i,1]]=mat[i]*mod[iloc[i,0]]
```

Sparse matrix multiplication

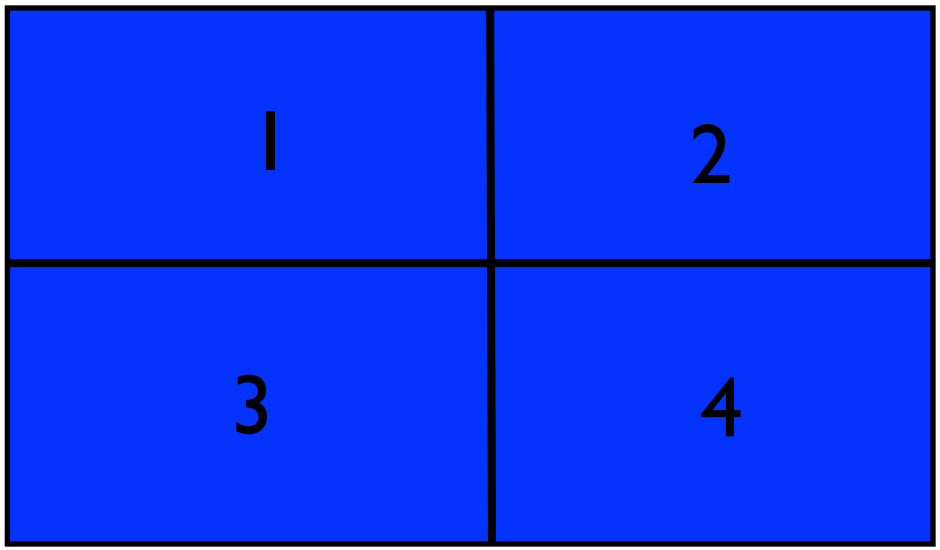


```
for i in range(npts):
        dat[iloc[i,1]]=mat[i]*mod[iloc[i,0]]
```

Embarrassingly parallel

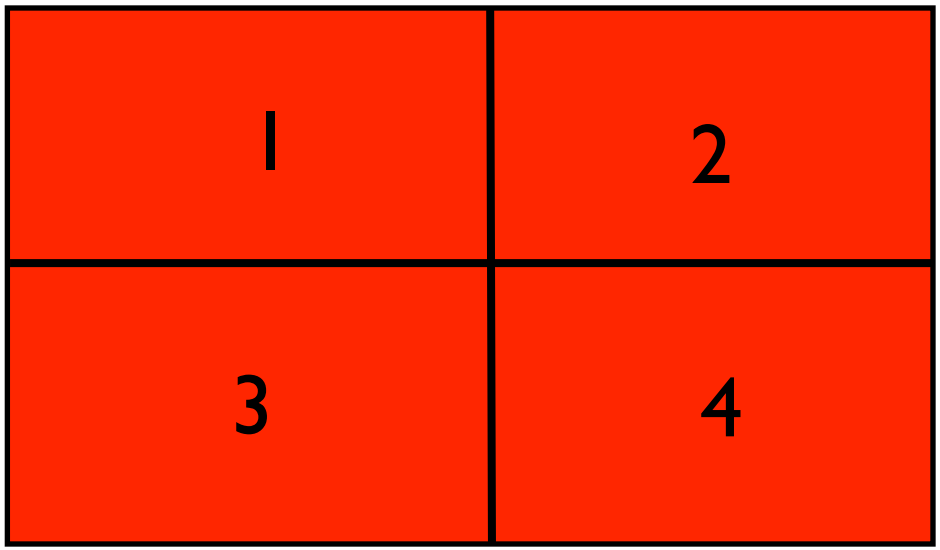
- Process 1 and 2 never need to share any information
- Problem scales to very large number of processors
- Everyone else is jealous

Domain decomposition



Does process 2 need to know anything about what process 1 computes?

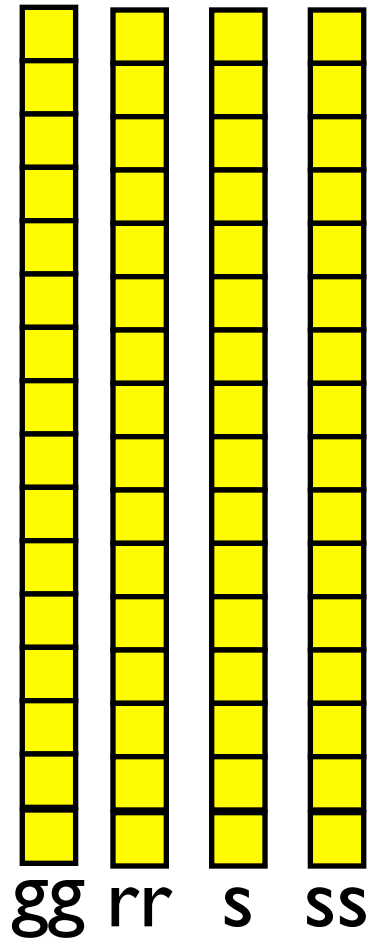
Functional decomposition



Does process 2 need information that process 1 calculates?

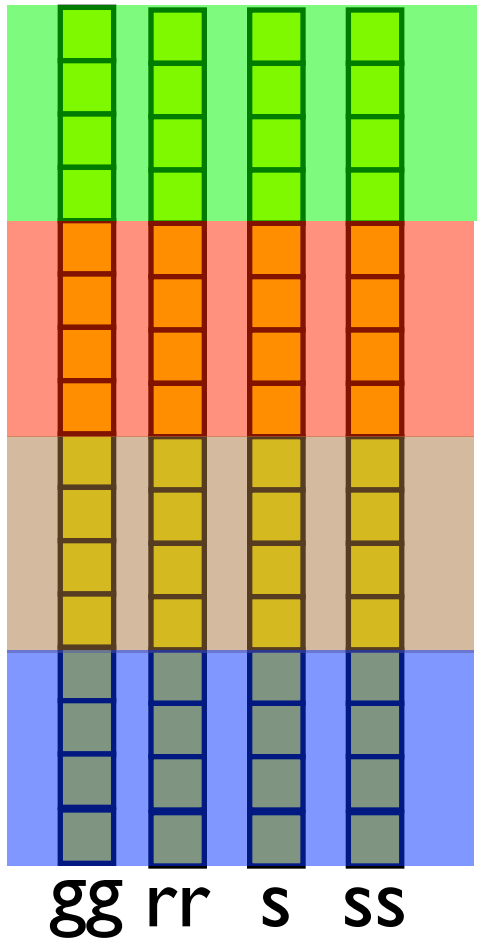
Vector operations

```
num=0
den=0
for i in range(gg.shape[0]):
    num+=gg[i]*rr[i]
for i in range(gg.shape[0]):
    den+=gg[i]*gg[i]
alfa=-num/den
```



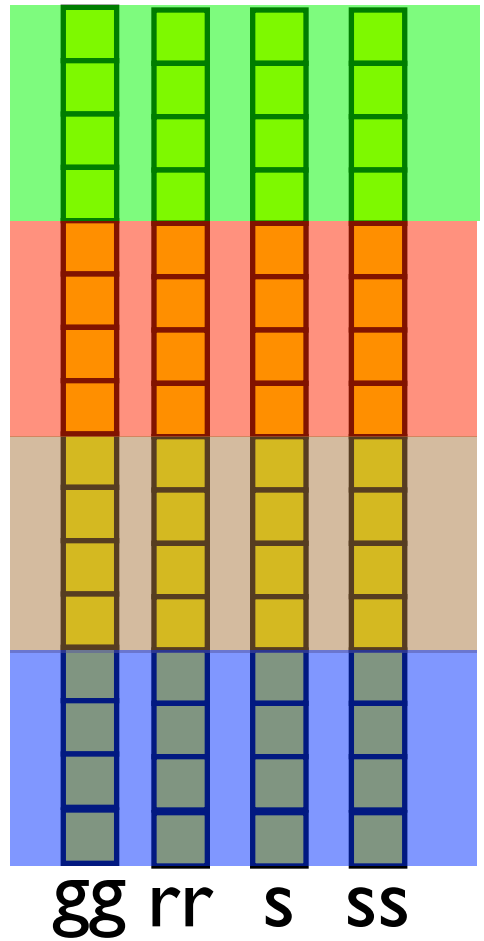
Vector operations

```
num=0
den=0
for i in range(gg.shape[0]):
    num+=gg[i]*rr[i]
for i in range(gg.shape[0]):
    den+=gg[i]*gg[i]
alfa=-num/den
```



Vector operations

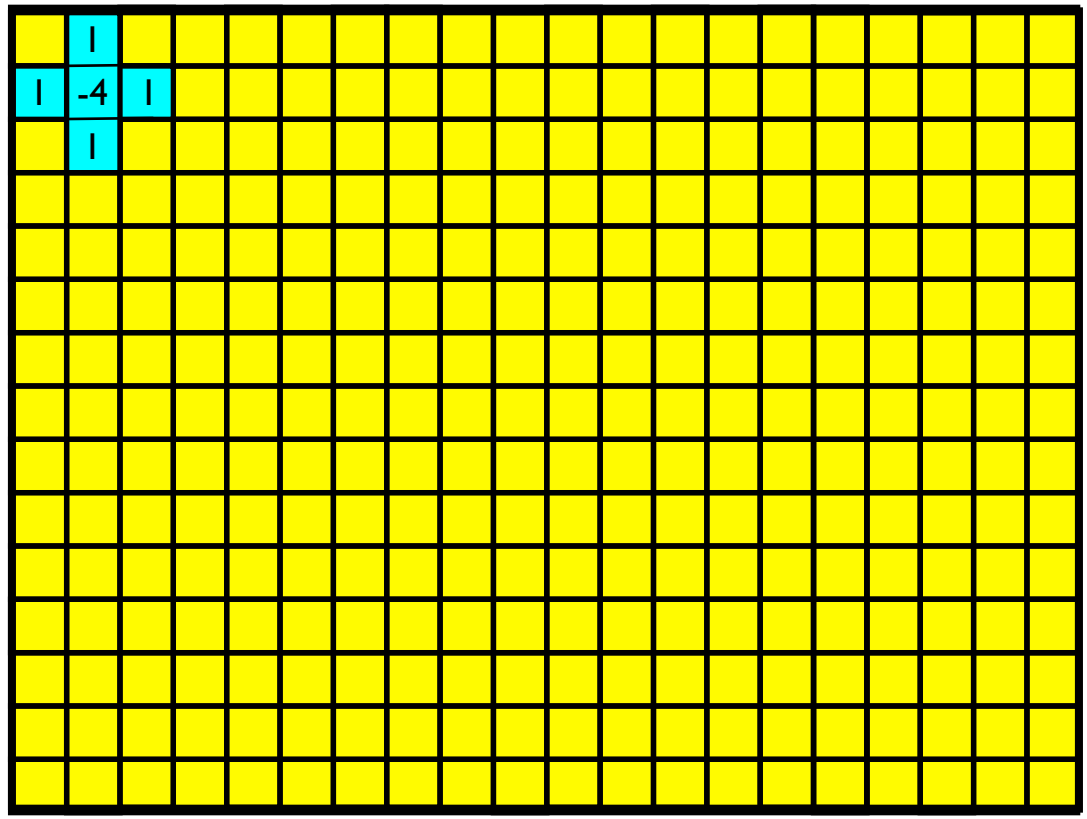
```
num=0
den=0
for i in range(gg.shape[0]):
    num+=gg[i]*rr[i]
for i in range(gg.shape[0]):
    den+=gg[i]*gg[i]
alfa=-num/den
```



Convolution

```
def timeStep(pm,pp,pn,dvv):
        convolve()
            ..
            ..
```

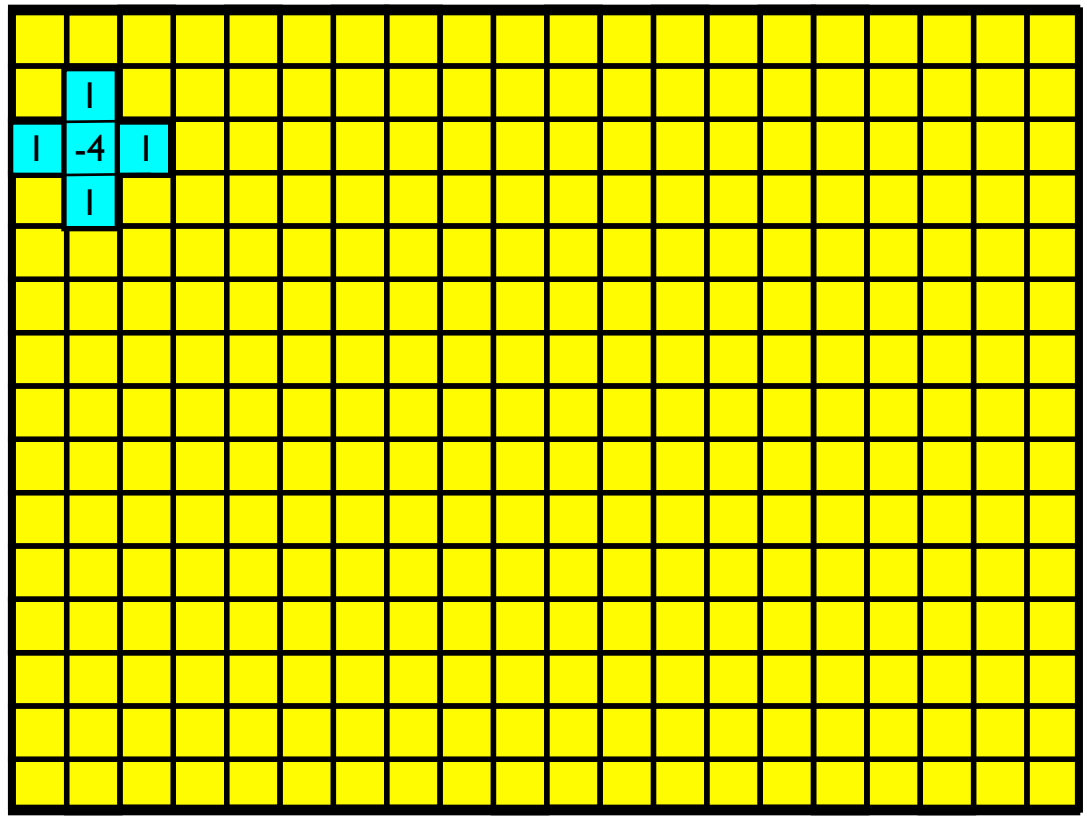
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```



Convolution

```
def timeStep(pm,pp,pn,dvv):
        convolve()
        ..
        ..
```

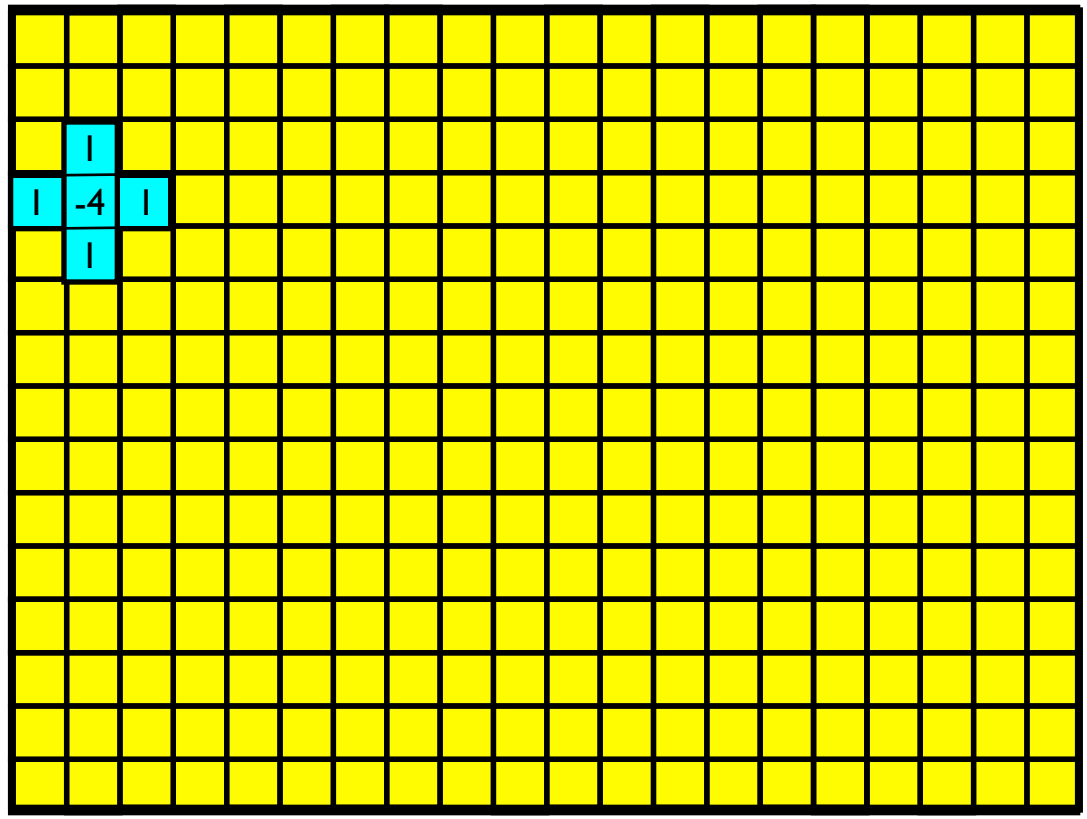
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```



Convolution

```
def timeStep(pm,pp,pn,dvv):
        convolve()
        ..
        ..
```

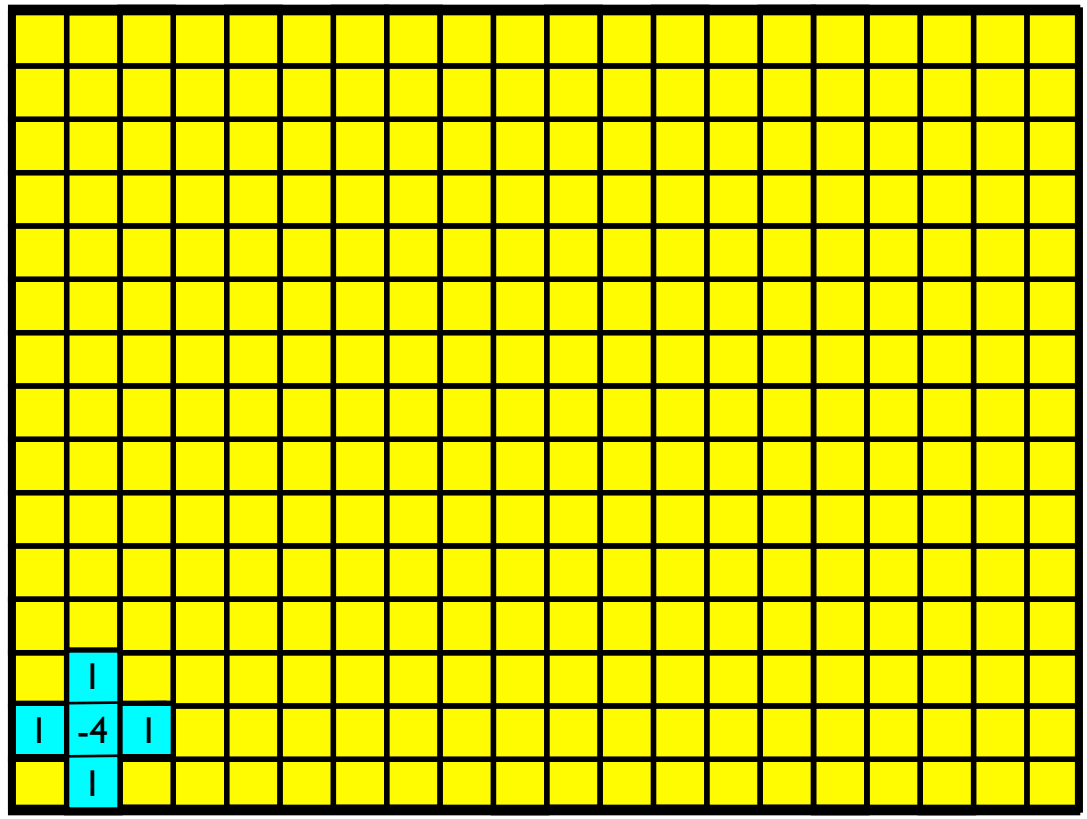
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```



Convolution

```
def timeStep(pm,pp,pn,dvv):
        convolve()
            ..
            ..
```

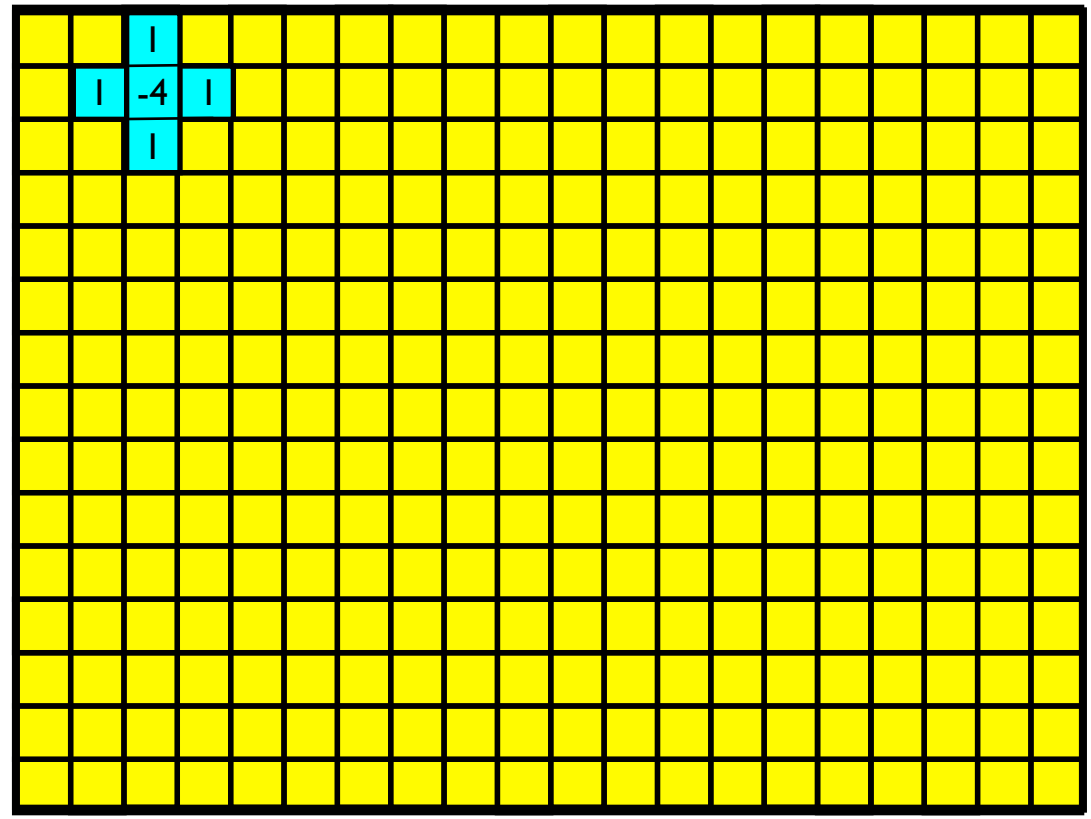
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```



Convolution

```
def timeStep(pm,pp,pn,dvv):
        convolve()
        ..
        ..
```

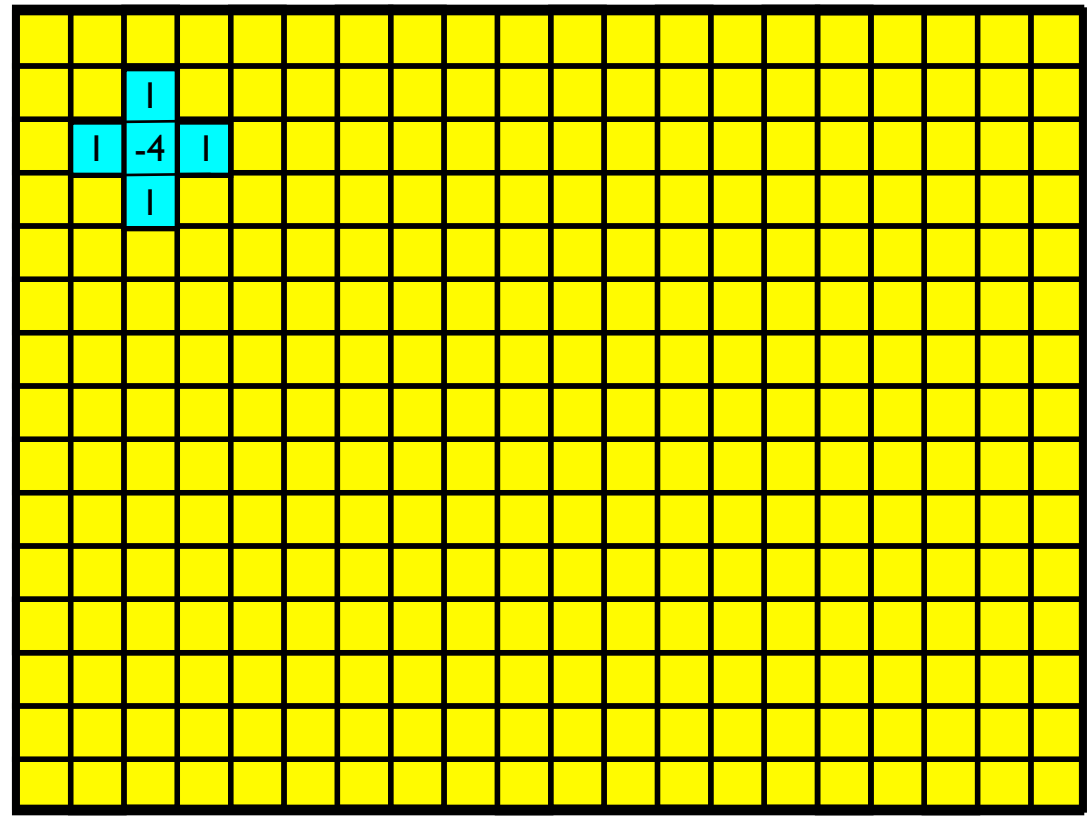
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```



Convolution

```
def timeStep(pm,pp,pn,dvv):
        convolve()
        ..
        ..
```

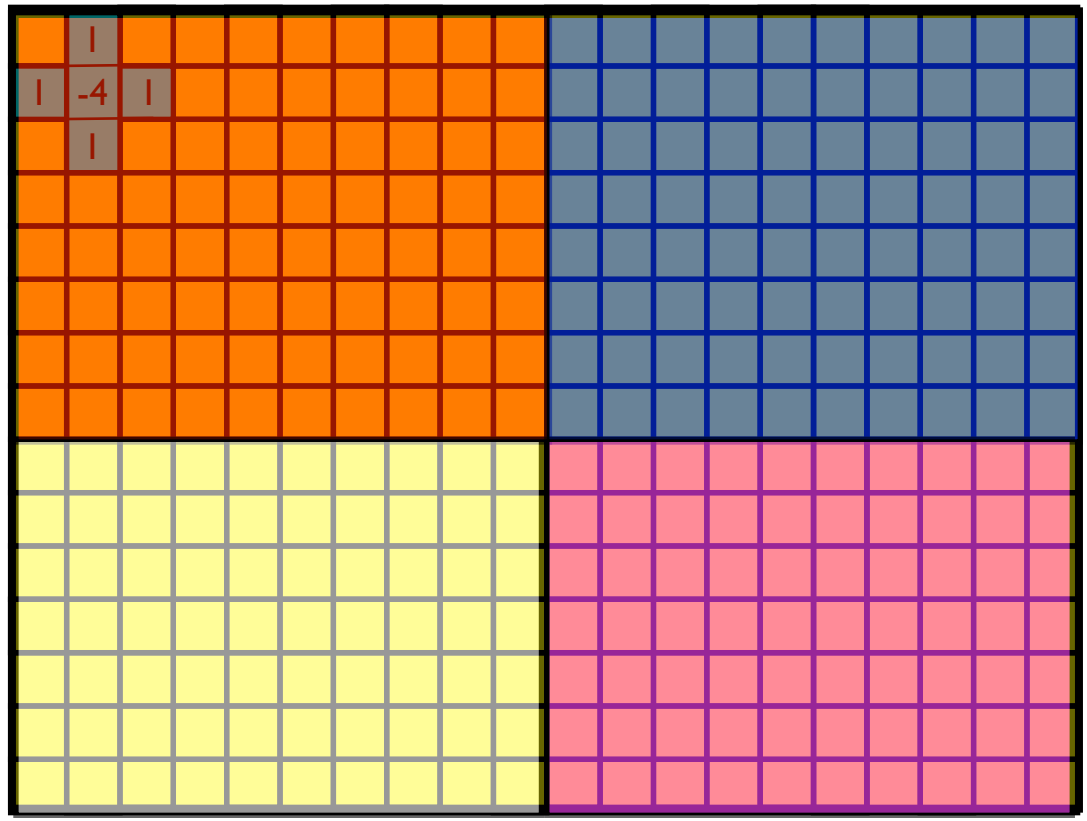
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```



Nodes split computational domain

```
def timeStep(pm,pp,pn,dvv):
        convolve()
            ..
            ..
```

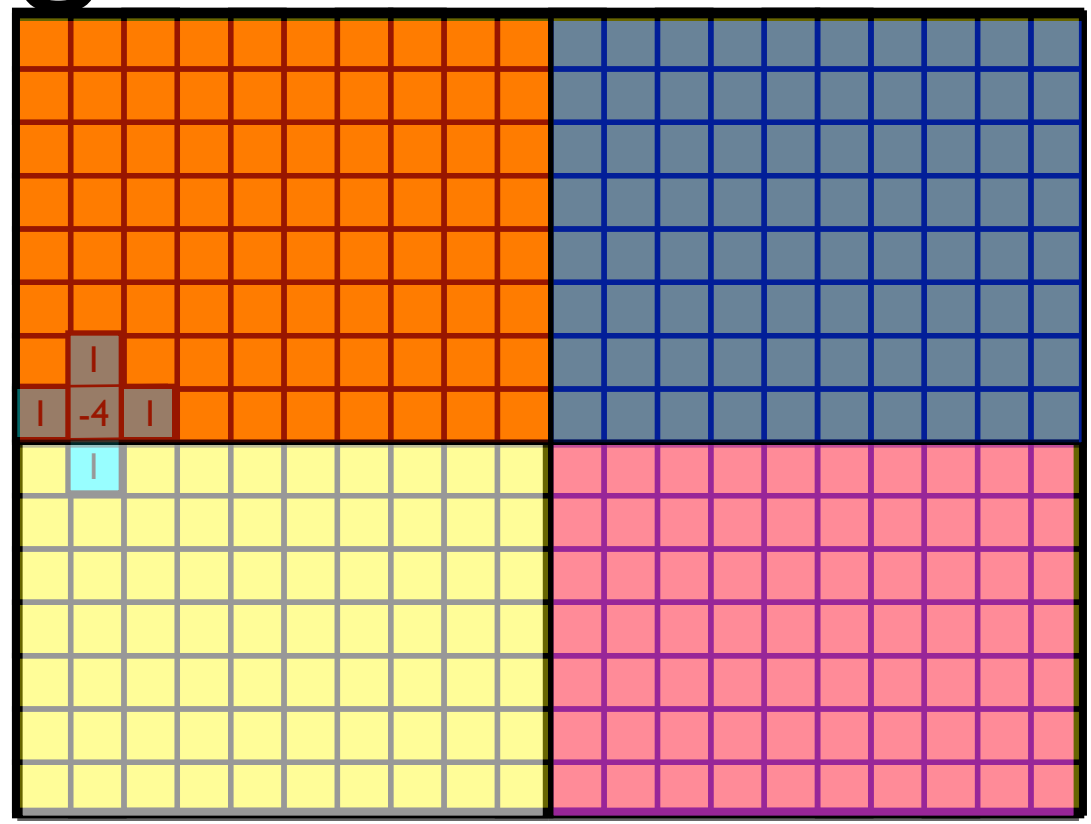
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```



Need to grab info from neighbors

```
def timeStep(pm,pp,pn,dvv):
        convolve()
            ..
            ..
```

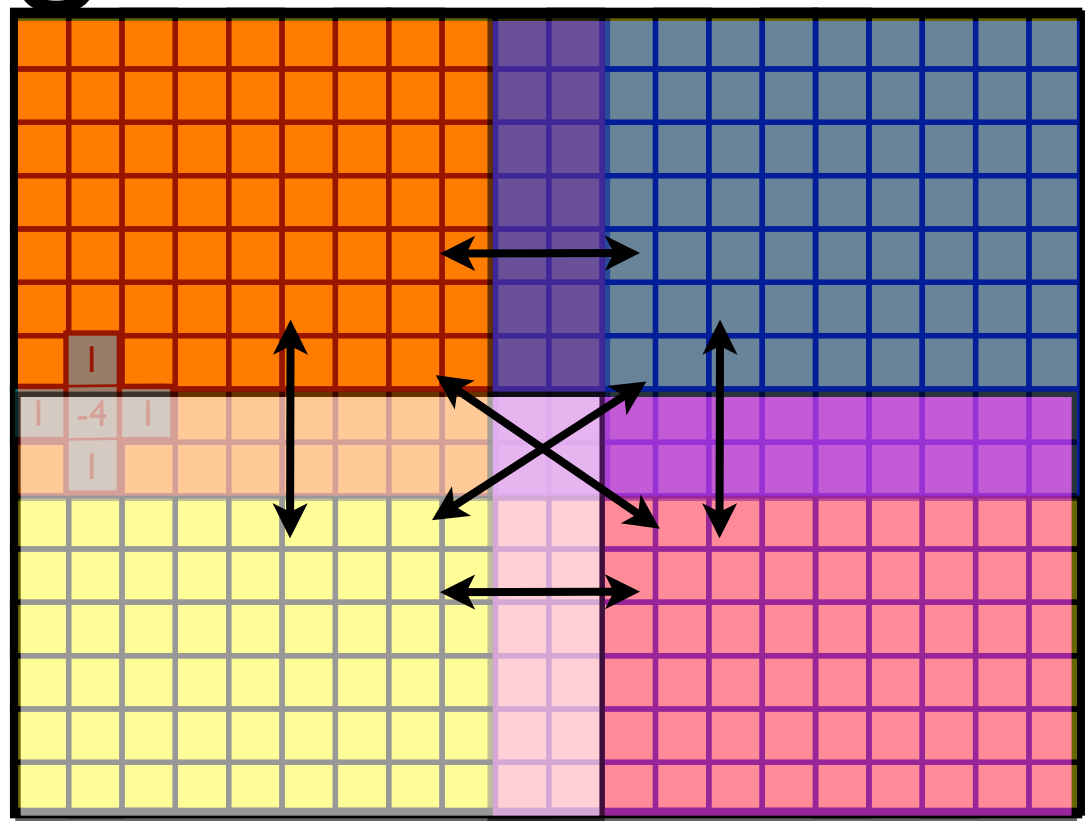
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```

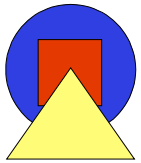


Need to grab info from neighbors

```
def timeStep(pm,pp,pn,dvv)
    convolve()
    …
    exchangeBoundaries()
```

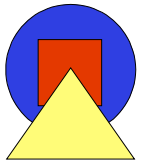
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```





Cost of Communication

- Communicating data takes time
  - Inter-task comm. has overhead
  - Often synchronization is necessary
- Communication is much more "expensive" than computation
  - Communicating data needs to save a lot of computation before it pays off
    - Infiniband needs < 10μs to set up communication
    - 1.2GHz AMD Athlon CPU needs ~0.8ns to perform one floating point operation (Flop)
    - 12,500 floating point operations per communication setup!

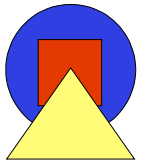


Cost of Communication

- Formula for the time needed to transmit data

$$\text{cost} = L + \frac{N}{B}$$

L = Latency [s]
N = number of bytes [byte]
B = Bandwidth [byte/s]
cost [s]



Communication Hardware

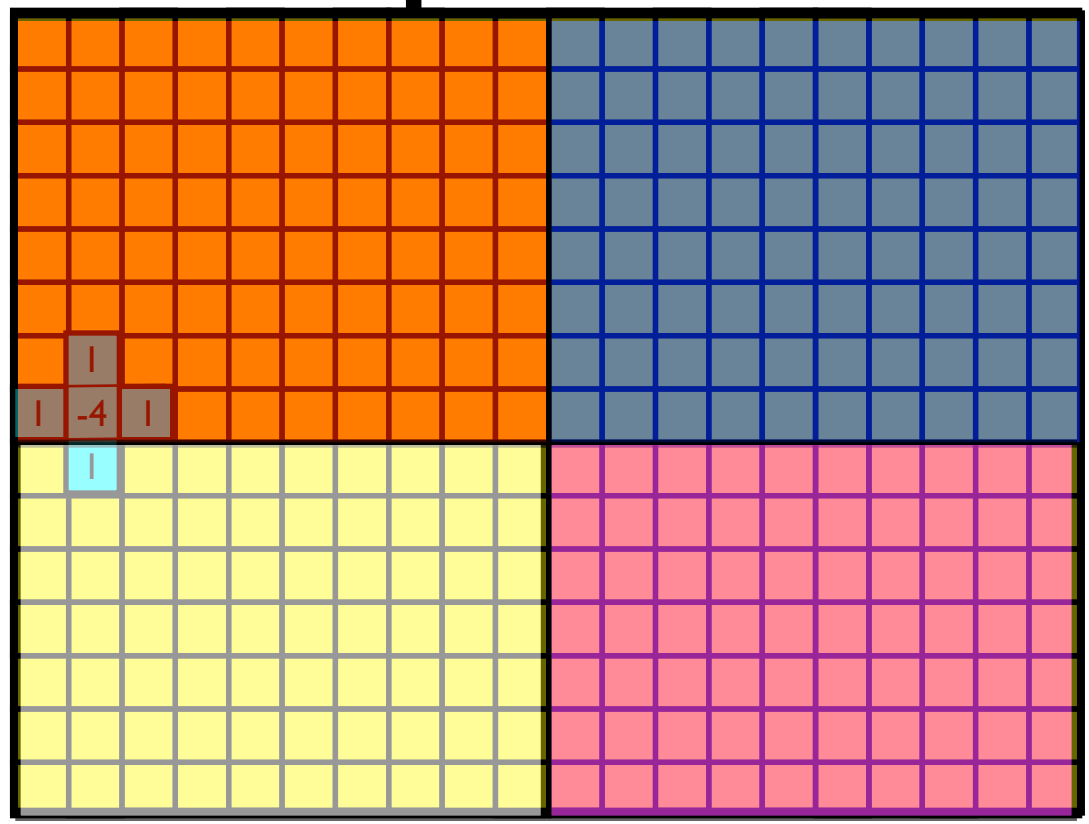
Architecture	Comment	Bandwidth	Latency
Myrinet http://www.myricom.com/	Proprietary but commodity	Sust. one-way for large messages: ~1.2GB/s	short messages: ~3μs
Infiniband http://www.infinibandta.org/	Vendor indep. standard	~900MB/s (4x HCAs)	~10μs
Qadrics (QsNet) http://www.quadrics.com/	Expensive, proprietary	~900MB/s	~2μs
Gigabit Ethernet	commodity	~100MB/s	~60μs

Custom: SGI, IBM, Cray, Sun, Compaq, ...

Amount of work vs amount of computation

```
def timeStep(pm,pp,pn,dvv)
    convolve()
    …
    exchangeBoundaries()
```

```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```



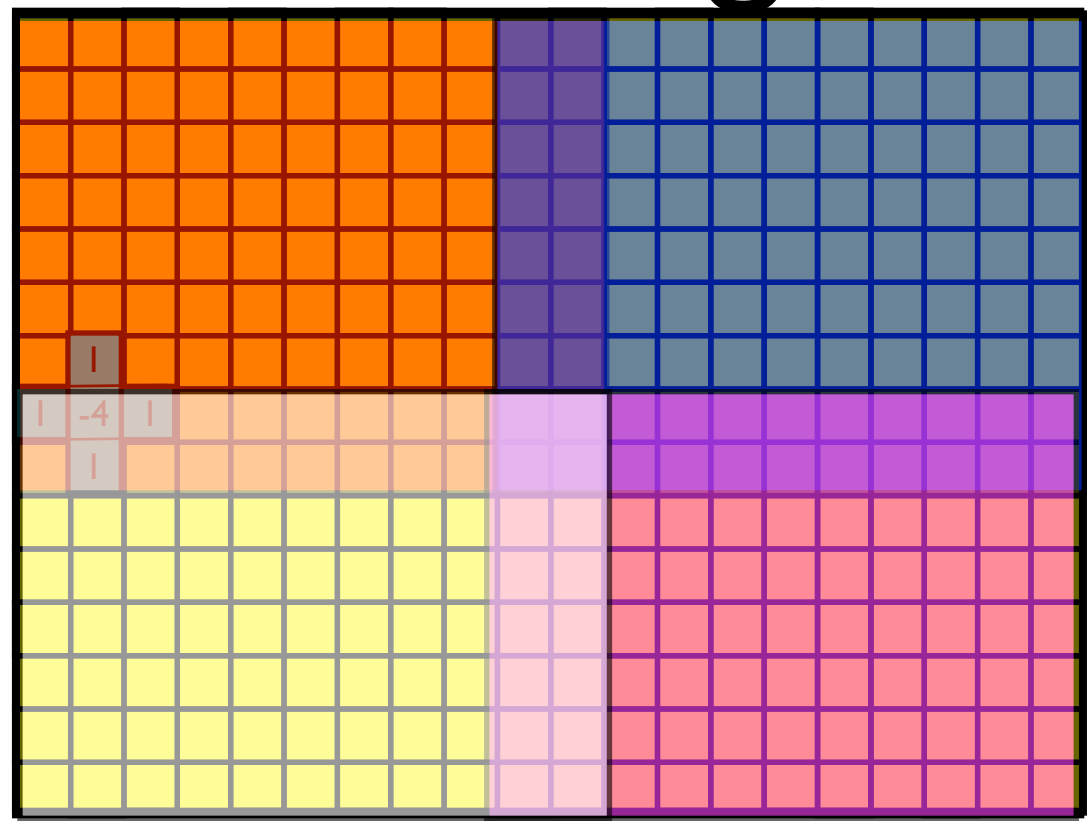
Granularity

- Ratio of computation to communication
- Fine-grain parallelism
  - Low computation to communication
- Coarse-grain parallelism
  - High computation to communication
  - Generally better

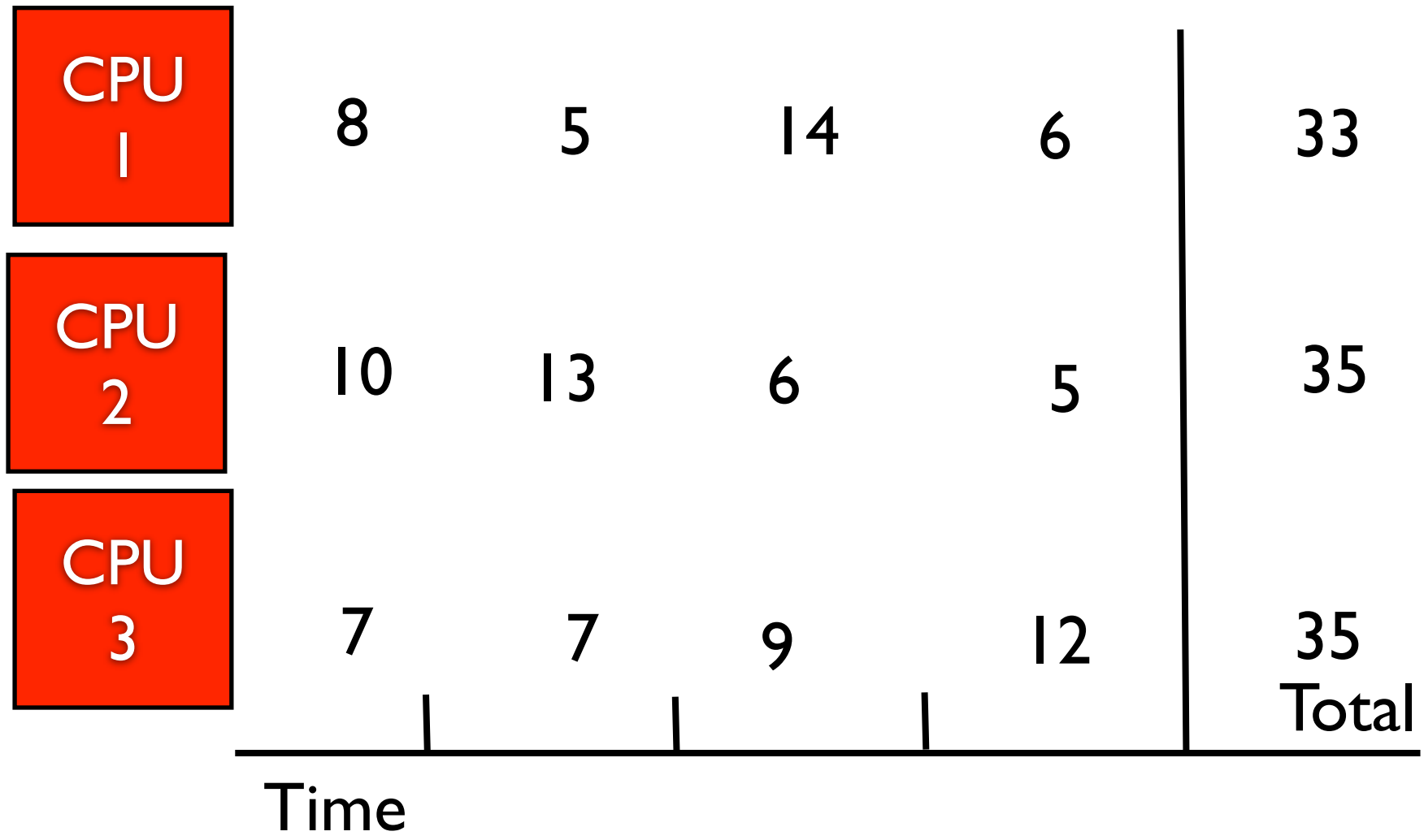
Both processes need to be at the same stage

```
def timeStep(pm,pp,pn,dvv)
    convolve()
    …
    exchangeBoundaries()
```

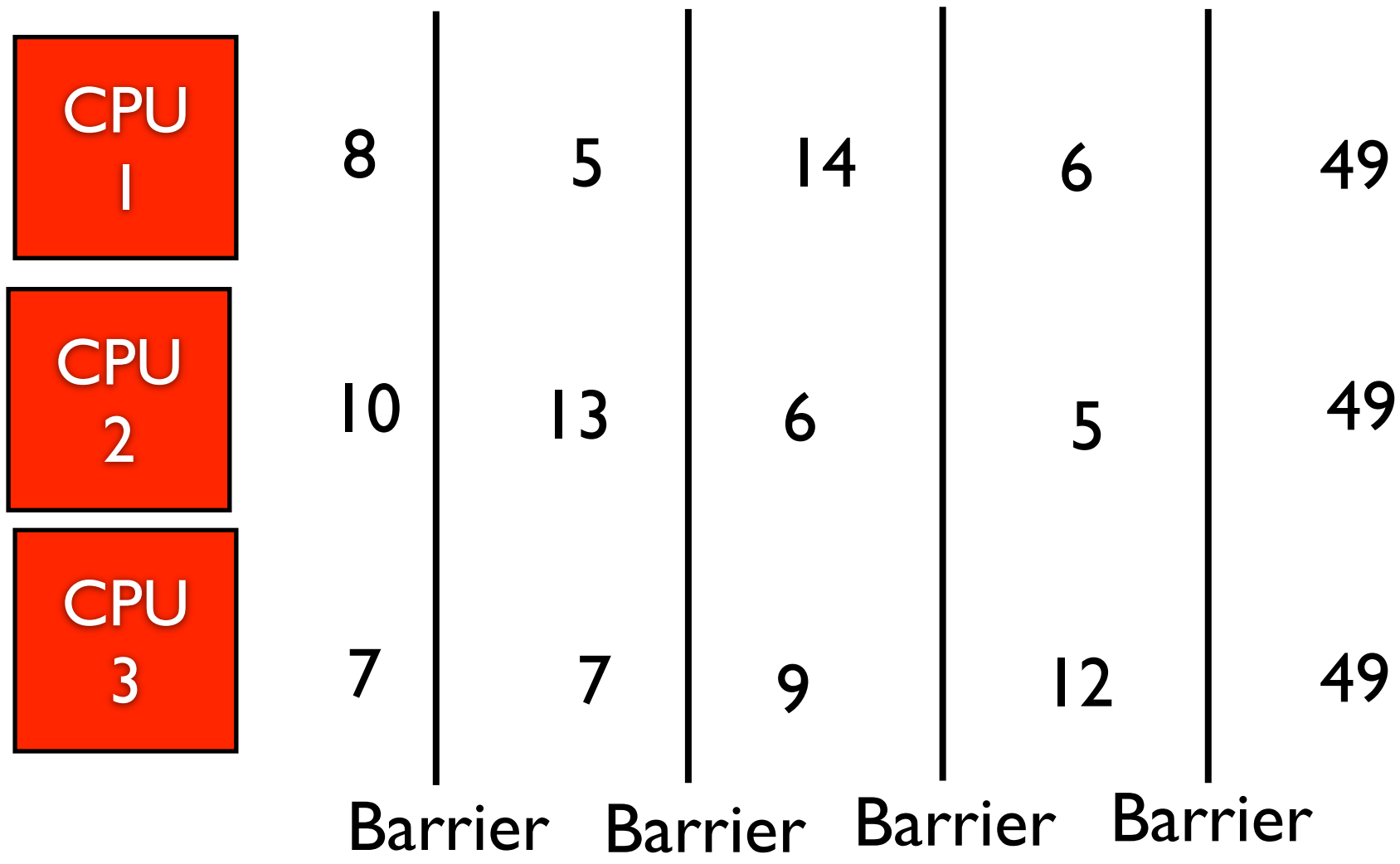
```
def convolve(pm,pp,dvv):
    for i2 in range(1,pp.shape[0]-1):
        for i1 in range(1,pp.shape[1]-1):
            pn[i2,i1]=dvv[i2,i1]*(-4*pp[i2,i1]+pp[i2-1,i1]+
              pp[i2+1,i1]+pp[i2,i1+1]+pp[i2,i1-1])
```



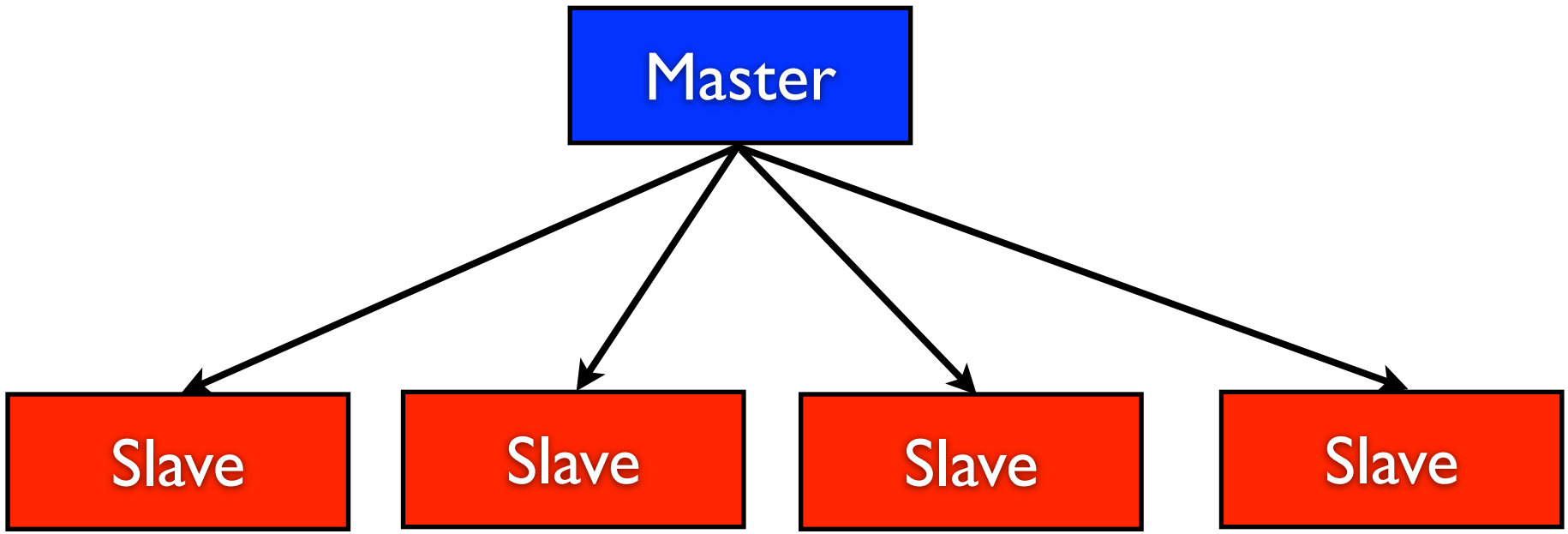
Cost of communicating Synchronization



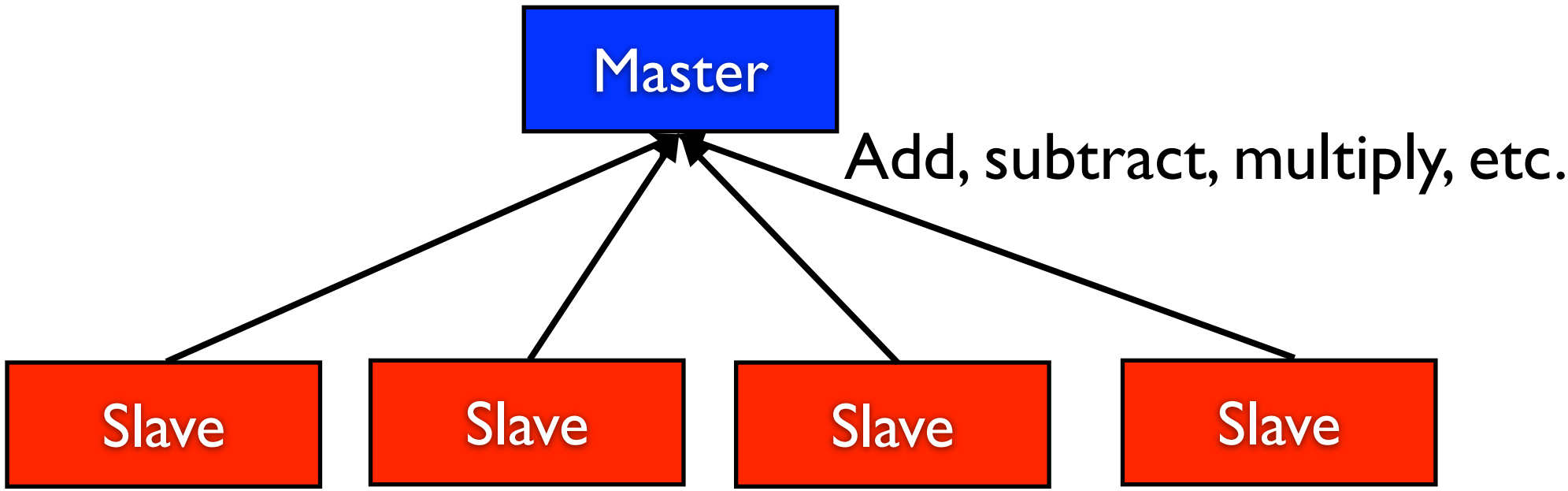
Cost of communicating Synchronization



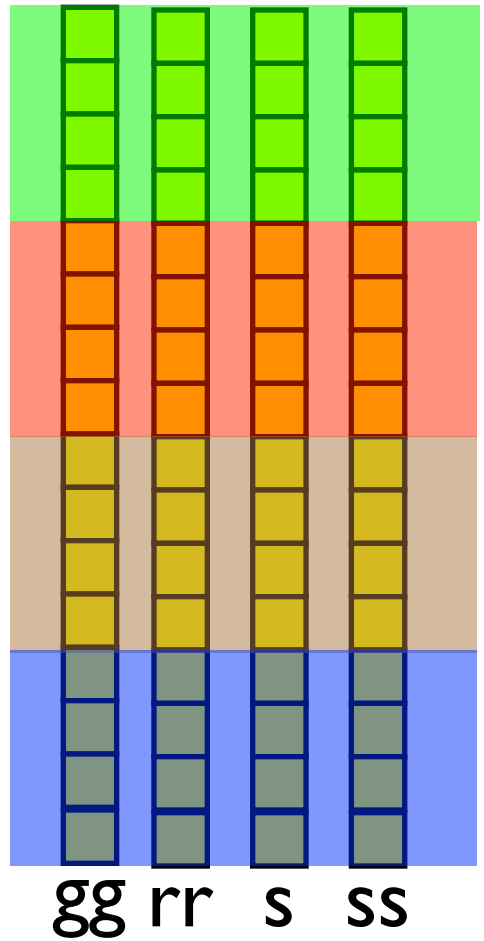
Collective communication: Broadcast



Collective communication: Reduce

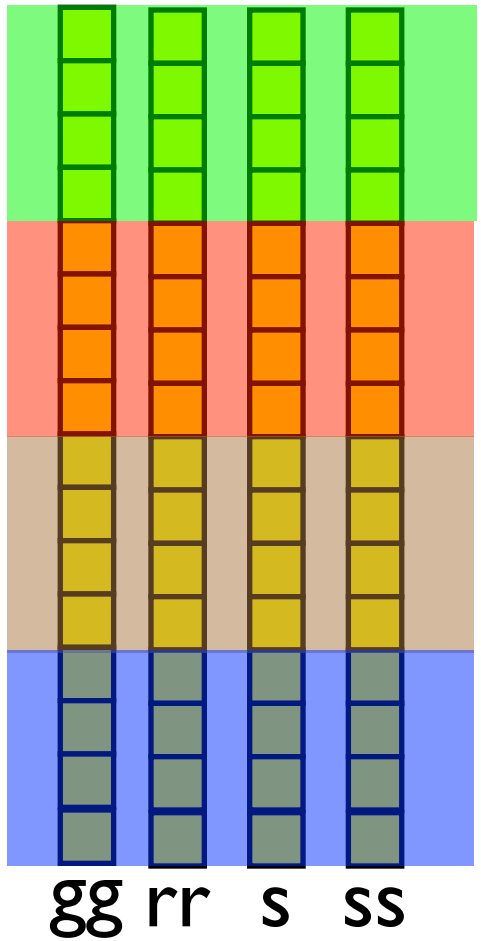


Vector operations



Collective communication

```
num=0
den=0
for i in range(gg.shape[0]):
    num+=gg[i]*rr[i]
for i in range(gg.shape[0]):
    den+=gg[i]*gg[i]
alfa=-num/den
    collect(den)
    collect(num)
    master:  alfa = - num/den
    broadcast alfa
```

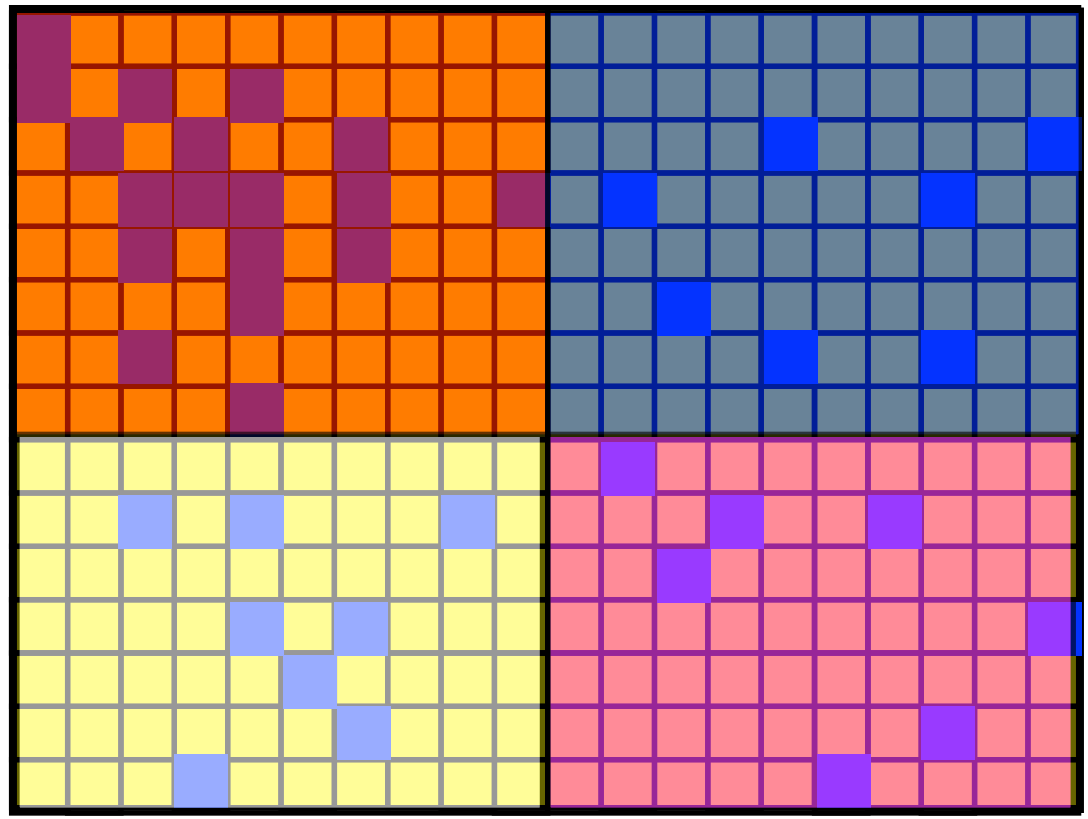


One node collects all local num,den

Calculates alfa

send to all processors alfa

Load balancing



Amount of work might vary significantly

```
for i in range(npts):
    dat[iloc[i,1]]=mat[i]*mod[iloc[i,0]]
```

Load balancing

```
Master:
while(all_blocks_not_done):
  send_block()
```

```
while(all_blocks_not_done):
  receive_block()
    for i in range(npts):
      dat[iloc[i,1]]=mat[i]*mod[iloc[i,0]]
```