Processor Architecture

Victor Eijkhout

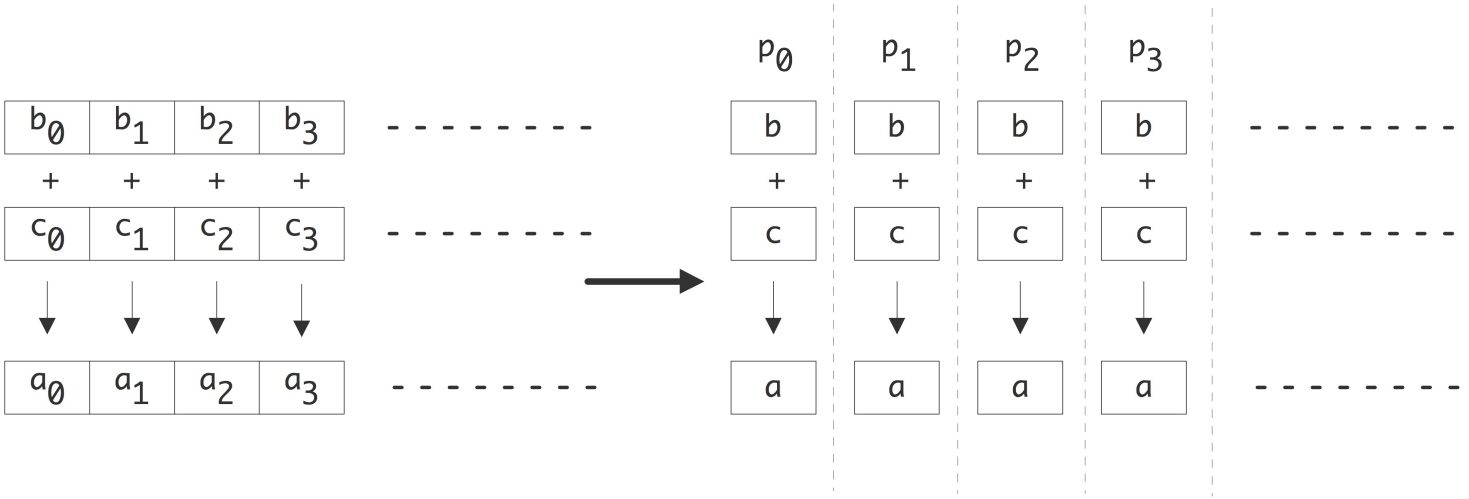
PCSE 2016

1. Basic concepts
2. Theoretical concepts
3. The SIMD/MIMD/SPMD/SIMT model for parallelism
4. Characterization of parallelism by memory model
5. Interconnects and topologies, theoretical concepts
6. Programming models
7. Load balancing, locality, space-filling curves

Table of Contents

The basic idea

```
for (i=0; i<n; i++)
  a[i] = b[i] + c[i];
```




```
for (i=my_low; i<my_high; i++)
  a[i] = b[i] + c[i];
```

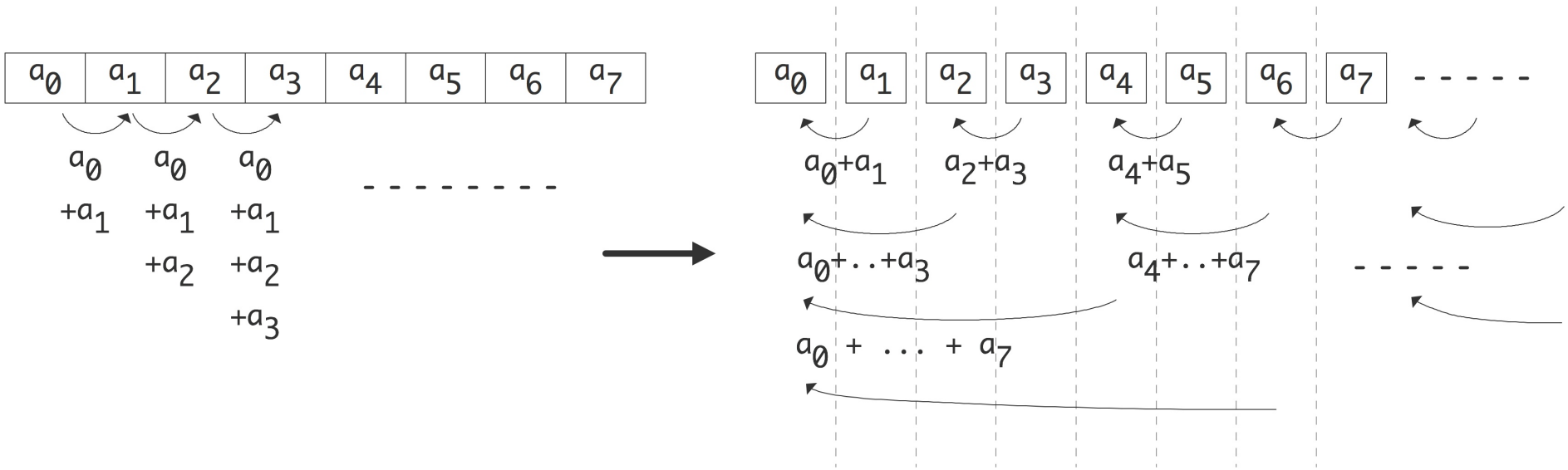
Time goes down linearly with processors

If it was always that easy. . .

```
s = 0;
for (i=0; i<n; i++)
  s += x[i]
```

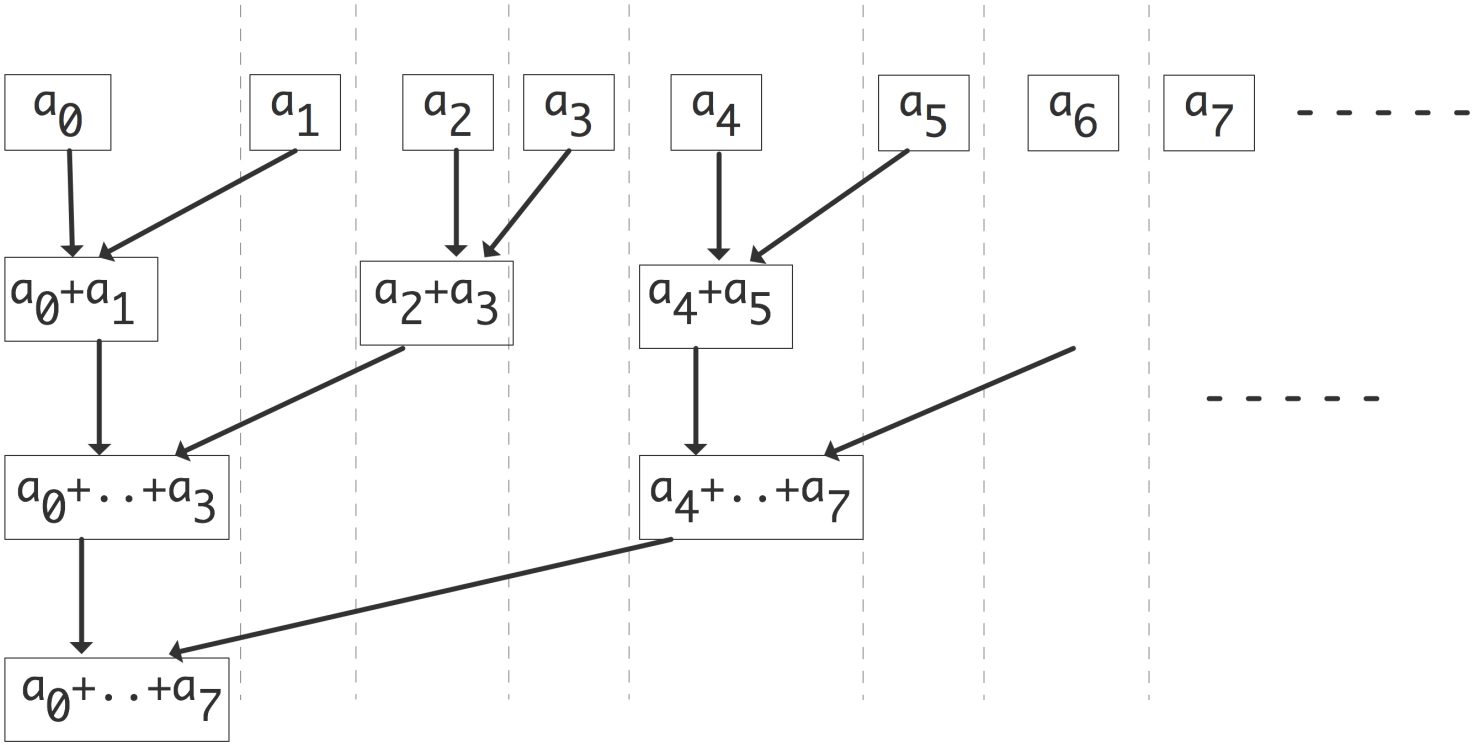
Recoding

```
for (s=2; s<n; s*=2)
  for (i=0; i<n; i+=s)
    x[i] += x[i+s/2]
```



And then there is hardware

Topology of the processors:



increasing distance: limit on parallel speedup

Table of Contents

Granularity

Definition

Definition: granularity is the measure for how many operations can be performed between synchronizations

Instruction level parallelism

$$a \leftarrow b + c$$
$$d \leftarrow e * f$$

For the compiler / processor to worry about

Data parallelism

```
for (i=0; i<1000000; i++)
  a[i] = 2*b[i];
```

- Array processors, vector instructions, pipelining, GPUs
- Sometimes harder to discover
- Often used mixed with other forms of parallelism

Task-level parallelism

if optimal *(root)* **then**
 | **exit**
else
 | **parallel**: **SearchInTree** (leftchild),**SearchInTree** (rightchild)
end

Procedure SearchInTree(root)

Unsynchronized tasks: fork-join

general scheduler

while *there are tasks left* **do**
 | wait until a processor becomes inactive;
 | spawn a new task on it
end

Conveniently parallel

Example: Mandelbrot set

Parameter sweep,
often best handled by external tools

Medium-grain parallelism

Mix of data parallel and task parallel

```
my_lower_bound = // some processor-dependent number
my_upper_bound = // some processor-dependent number
for (i=my_lower_bound; i<my_upper_bound; i++)
  // the loop body goes here
```

Efficiency and scaling

Speedup

- Single processor time T_1, on p processors T_p
- speedup is $S_p = T_1/T_p$, $S_P \leq p$
- efficiency is $E_p = S_p/p$, $0 < E_p \leq 1$

Many caveats

- Is T_1 based on the same algorithm? The parallel code?
- Sometimes superlinear speedup.
- Can the problem be run on a single processor?
- Can the problem be evenly divided?

Limits on speedup/efficiency

- F_s sequential fraction, F_p parallelizable fraction
- $F_s + F_p = 1$
- $T_1 = (F_s + F_p)T_1 = F_s T_1 + F_p T_1$
- Amdahl's law: $T_p = F_s T_1 + F_p T_1/p$
- $P \to \infty$: $T_P \downarrow T_1 F_s$
- Speedup is limited by $S_P \leq 1/F_s$, efficiency is a decreasing function $E \sim 1/P$.
- loglog plot: straigth line with slope -1

Amdahl's law with communication overhead

- Communication independent of p: $T_p = T_1(F_s + F_p/P) + T_c$
- assume fully parallelizable: $F_p = 1$
- then $S_p = \frac{T_1}{T_1/p + T_c}$
- For reasonable speedup: $T_c \ll T_1/p$ or $p \ll T_1/T_c$:
  number of processors limited by ratio of scalar execution time and communication overhead

Gustafson's law

- Let $T_p = F_s + F_p \equiv 1$
- then $T_1 = F_s + p \cdot F_p$
- Speedup:

$$S_p = \frac{T_1}{T_p} = \frac{F_s + p \cdot F_p}{F_s + F_p} = F_s + p \cdot F_p = p - (p-1) \cdot F_s.$$

slowly decreasing function of p

Scaling

- Amdahl's law: strong scaling
  same problem over increasing processors
- Often more realistic: weak scaling
  increase problem size with number of processors,
  for instance keeping memory constant
- Weak scaling: $E_p > c$
- example (below): dense linear algebra

Simulation scaling

- Assumption: simulated time S, running time T constant, now increase precision
- m memory per processor, and P the number of processors

$$M = Pm \qquad \text{total memory.}$$

d the number of space dimensions of the problem, typically 2 or 3,

$$\Delta x = 1/M^{1/d} \qquad \text{grid spacing.}$$

- stability:

$$\Delta t = \begin{cases} \Delta x = 1 \,/\, M^{1/d} & \text{hyperbolic case} \\ \Delta x^2 = 1 \,/\, M^{2/d} & \text{parabolic case} \end{cases}$$

With a simulated time S:

$$k = S/\Delta t \qquad \text{time steps.}$$

- Assume time steps parallelizable

$$T = kM/P = \frac{S}{\Delta t} m.$$

Setting $T/S = C$, we find

$$m = C\Delta t,$$

memory per processor goes down.

$$m = C\Delta t = c \begin{cases} 1 \,/\, M^{1/d} & \text{hyperbolic case} \\ 1 \,/\, M^{2/d} & \text{parabolic case} \end{cases}$$

- Substituting $M = Pm$, we find ultimately

$$m = C \begin{cases} 1 \,/\, P^{1/(d+1)} & \text{hyperbolic} \\ 1 \,/\, P^{2/(d+2)} & \text{parabolic} \end{cases}$$

Table of Contents

Flynn Taxonomy

Consider instruction stream and data stream:

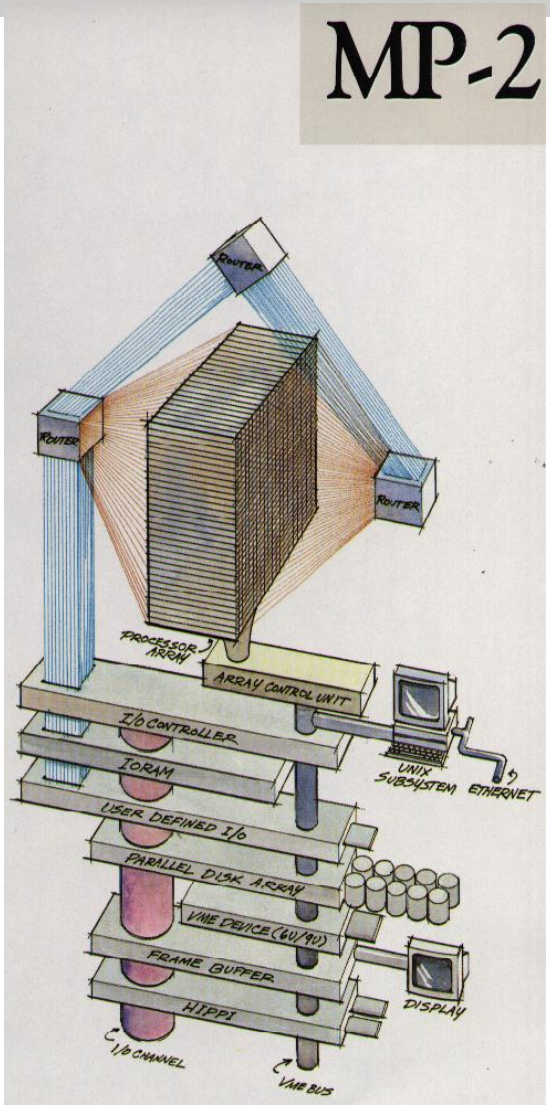
- SISD: single instruction single data
  used to be single processor, now single core
- MISD: multiple instruction single data
  redundant computing for fault tolerance?
- SIMD: single instruction multiple data
  data parallelism, pipelining, array processing, vector instructions
- MIMD: multiple instruction multiple data
  independent processors, clusters, MPPs

SIMD

- Relies on streams of identical operations
- See pipelining
- Recurrences hard to accomodate

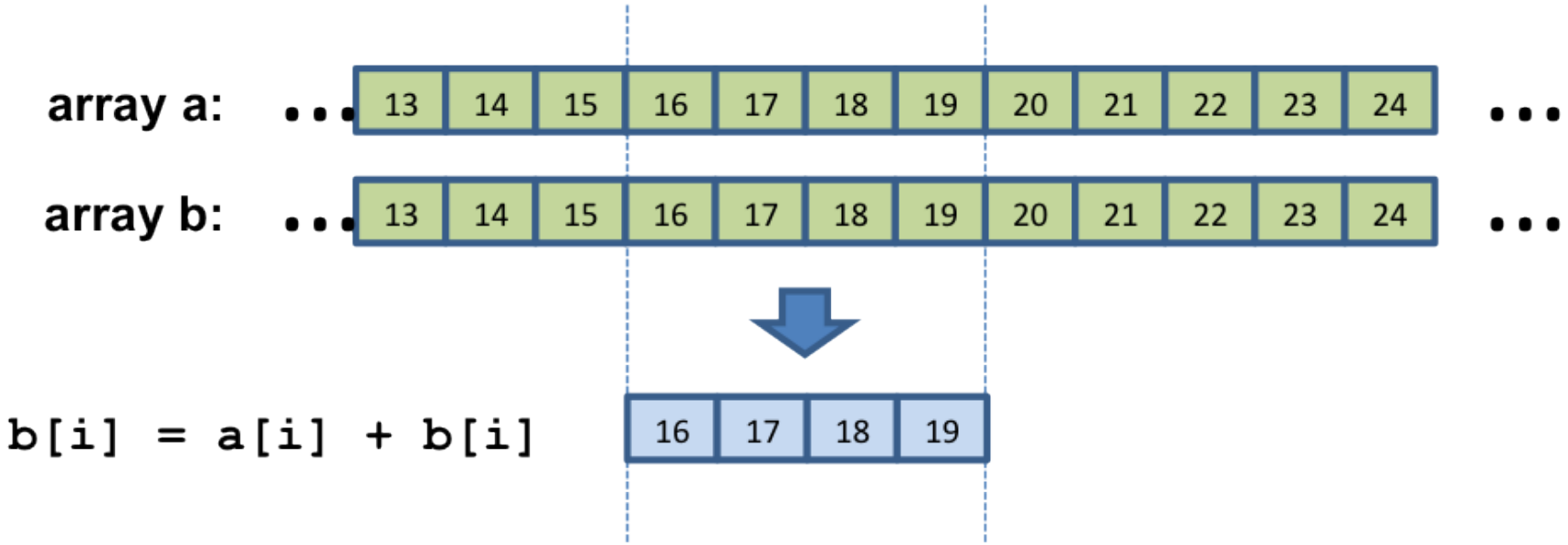
SIMD: array processors

Technology going back to the 1980s:
FPS, MasPar, CM, GoodYear
Major advantage: simplification of processor



SIMD as vector instructions

- Register width multiple of 8 bytes:
- simultaneous processing of more than one operand pair
- SSE: 2 operands,
- AVX: 4 or 8 operands



Controlling vector instructions

```
void func(float *restrict c, float *restrict a,
          float *restrict b, int n)
{
#pragma vector always
  for (int i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

This needs aligned data (`posix_memalign`)

New branches in the taxonomy

- SPMD: single program multiple data
  the way clusters are actually used
- SIMT: single instruction multiple threads
  the GPU model

MIMD becomes SPMD

- MIMD: independent processors, independent instruction streams, independent data
- In practice very little true independence: usally the same executable Single Program Multiple Data
- Exceptional example: climate codes
- Old-style SPMD: cluster of single-processor nodes
- New-style: cluster of multicore nodes, ignore shared caches / memory
- (We'll get to hybrid computing in a minute)

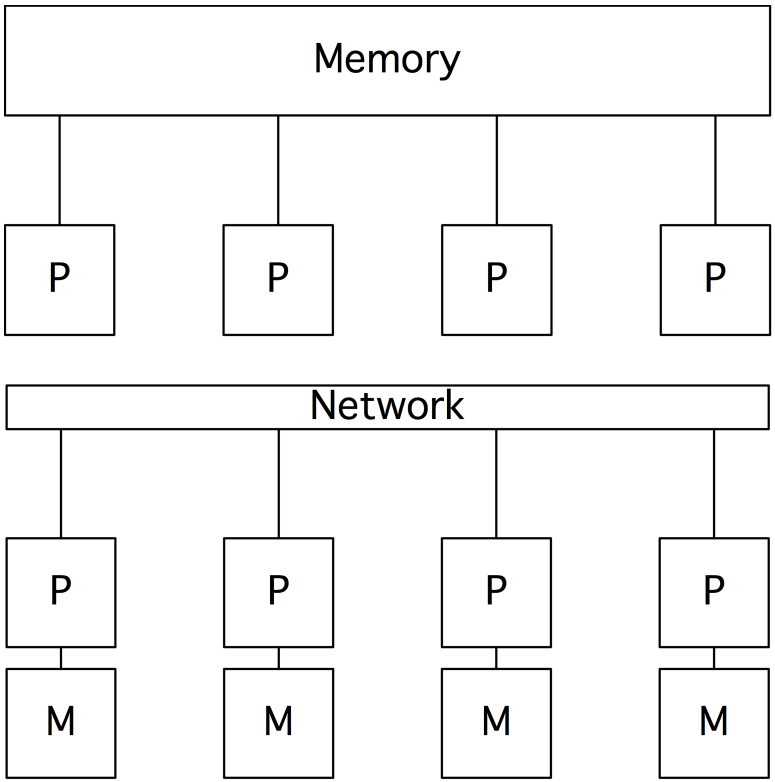
GPUs and data paralleism

Lockstep in thread block,
single instruction model between streaming processors

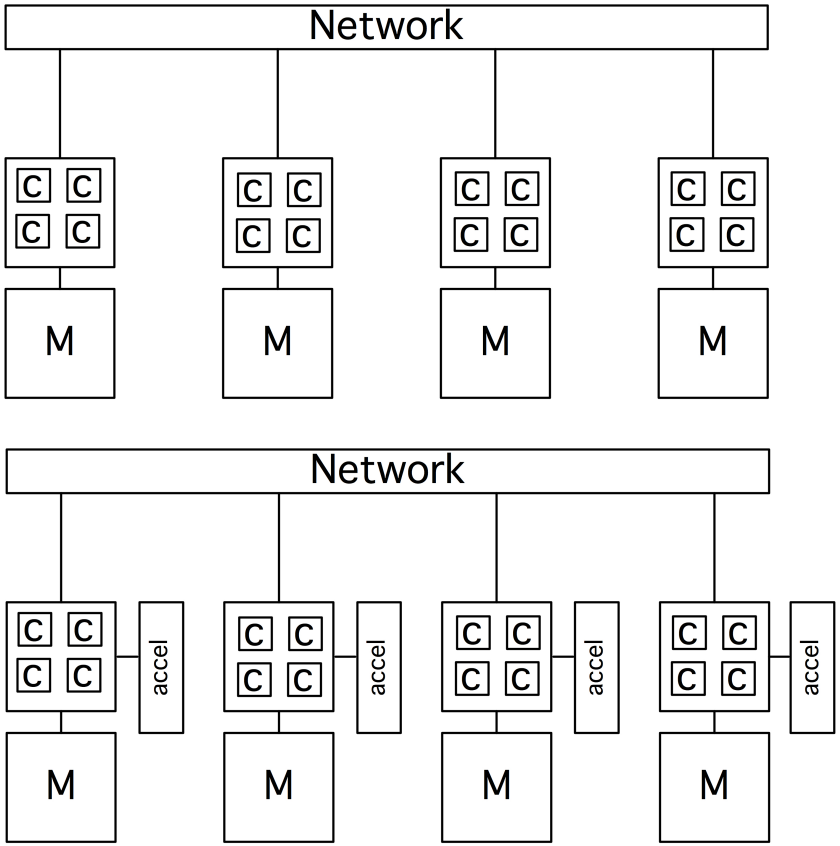
(more about GPU threads later)

Table of Contents

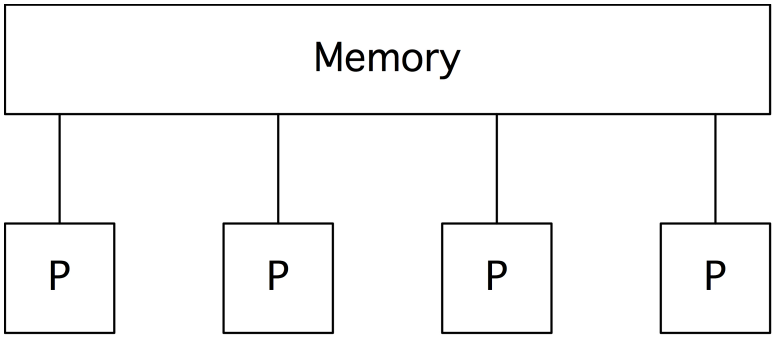
Major types of memory organization, classic



Major types of memory organization, contemporary



Symmetric multi-processing



- The ideal case of shared memory:
  every address equally accessible
- This hasn't existed in a while
  (Tim Mattson claims Cray-2)
- Danger signs: shared memory programming pretends that memory access is symmetric
  in fact: hides reality from you

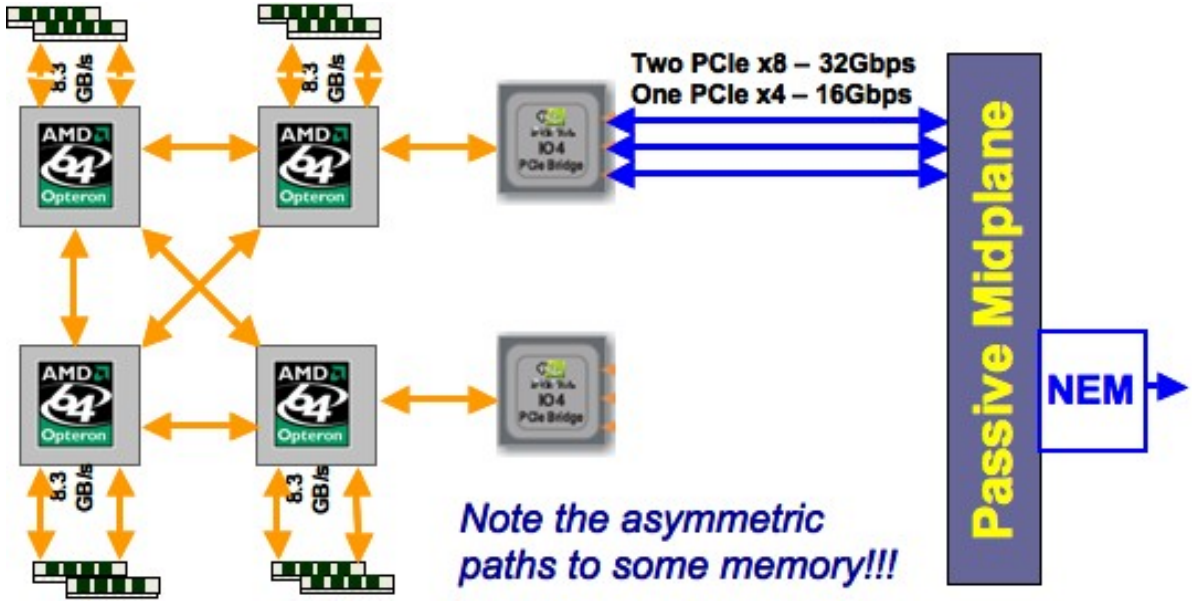
SMP, bus design

- Bus: all processors on the same wires to memory
- Not very scalable: requires slow processors or cache memory
- Cache coherence easy by 'snooping'

Non-uniform Memory Access

Memory is equally programmable, but not equally accessible

- Different caches, different affinity



- Distributed shared memory: network latency
  ScaleMP and other products watch me not believe it

Picture of NUMA

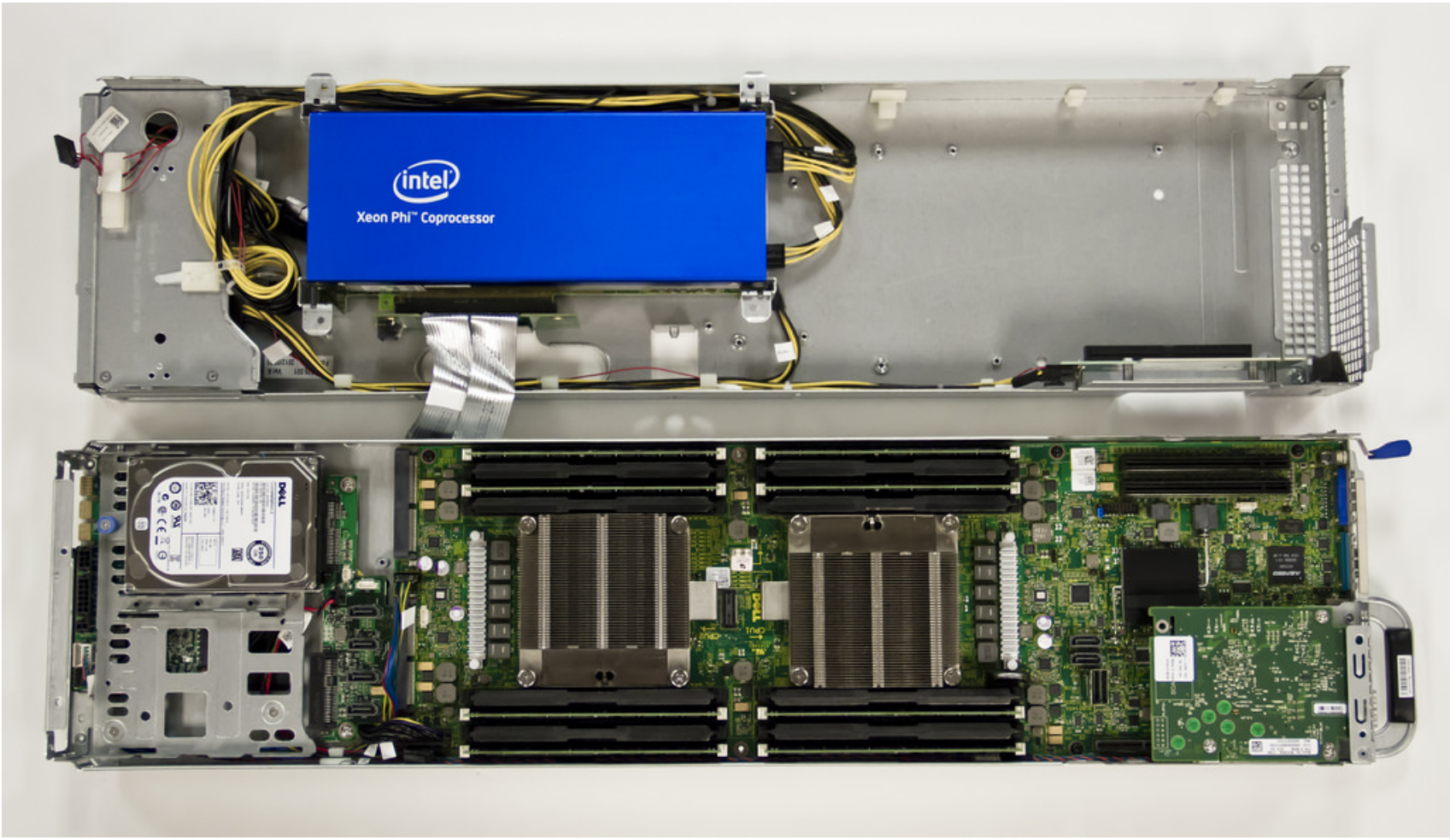


Table of Contents

Topology concepts

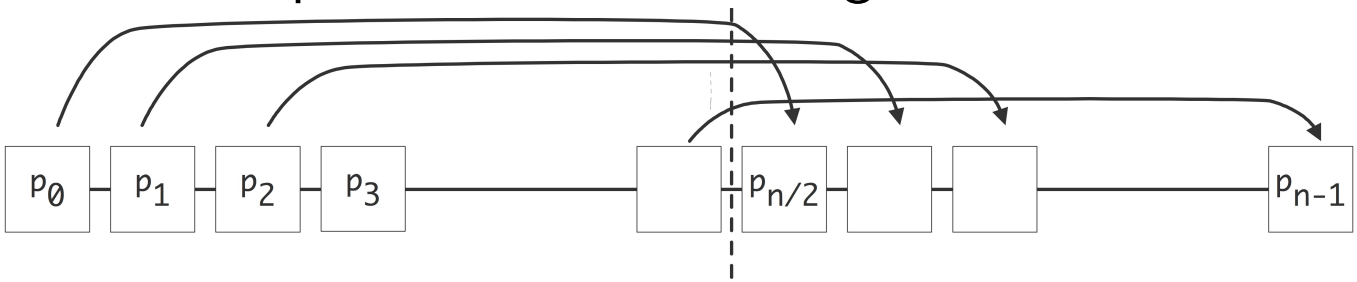
- Hardware characteristics
- Software requirement
- Design: how ‘close’ are processors?

Graph theory

- Degree: number of connections from one processor to others
- Diameter: maximum minimum distance (measured in hops)

Bandwidth

- Bandwidth per wire is nice, adding over all wires is nice, but. . .



- Bisection width: minimum number of wires through a cut
- Bisection bandwidth: bandwidth through a bisection

Design 1: bus

Already discussed; simple design, does not scale very far

Design 2: linear arrays

- Degree 2, diameter P, bisection width 1
- Scales nicely!
- but low bisection width

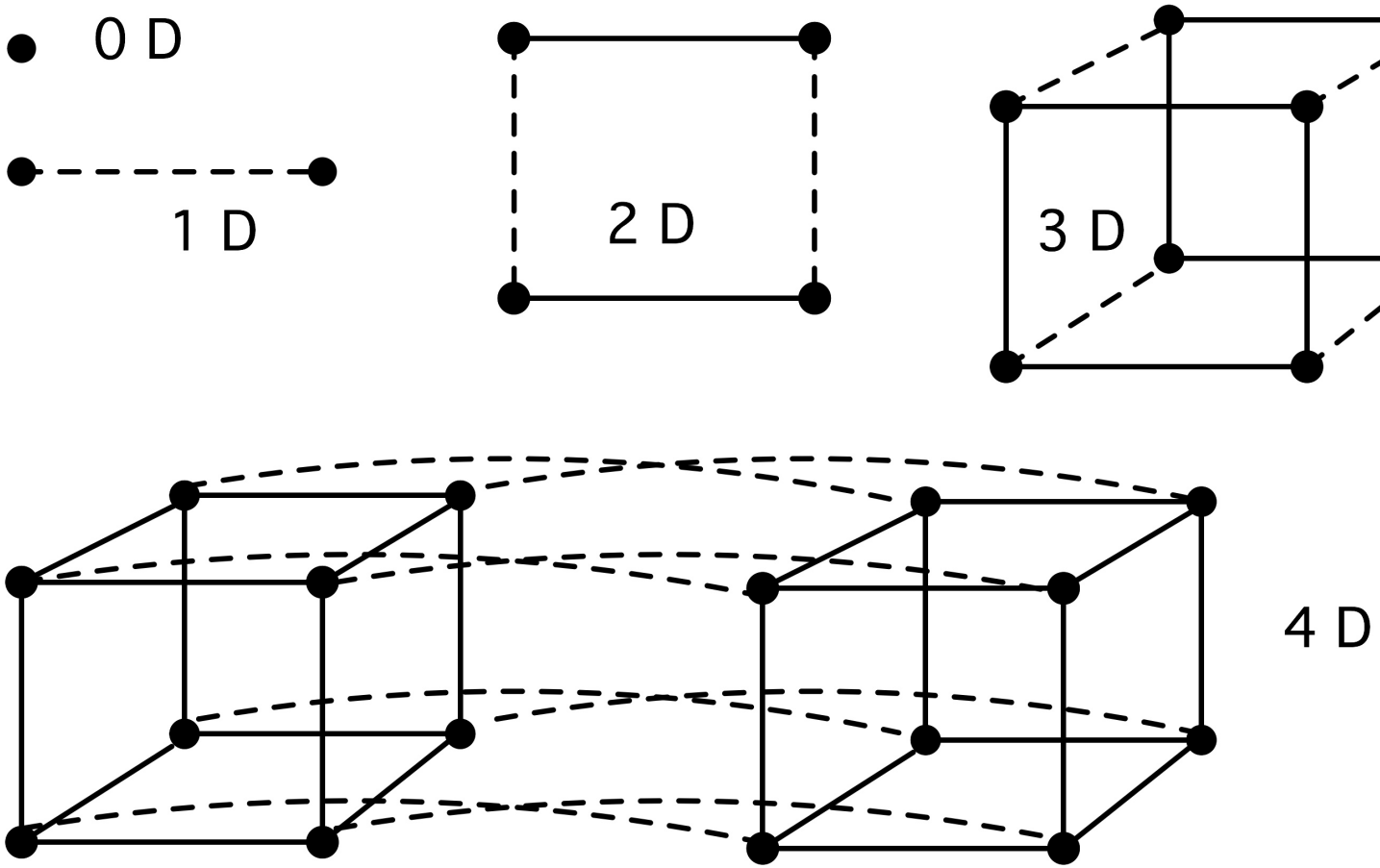
Exercise

Flip last bit, flip one before, . . .

Design 3: 2/3-D arrays

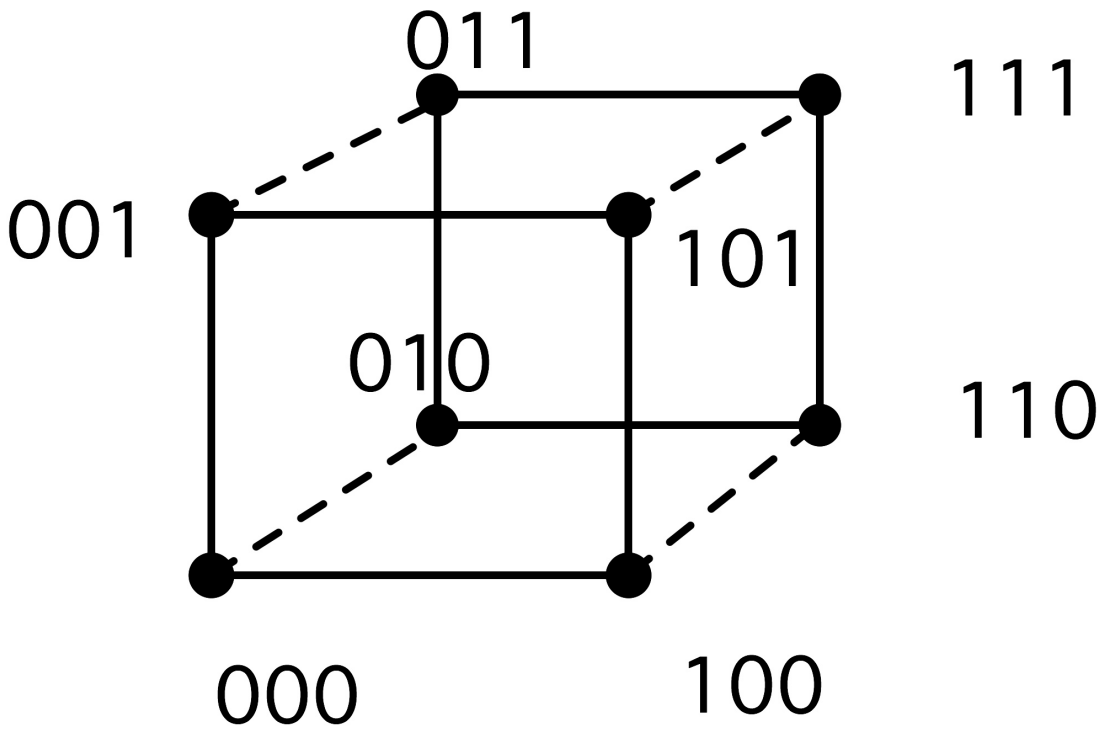
- Degree $2d$, diameter $P^{1/d}$
- Natural design: nature is three-dimensional
- More dimensions: less contention.
  K-machine is 6-dimensional

Design 3: Hypercubes



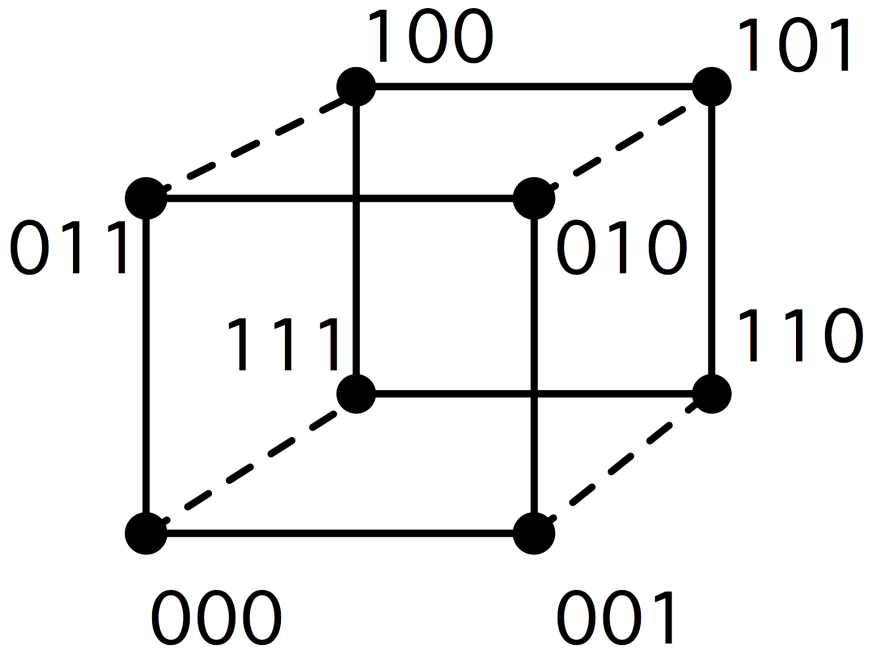
Hypercube numbering

Naive numbering:



Gray codes

Embedding linear numbering in hypercube:



Binary reflected Gray code

1D Gray code:

					0	1			

2D Gray code:

1D code and reflection:					0	1	⋮	1	0
append 0 and 1 bit:					0	0	⋮	1	1

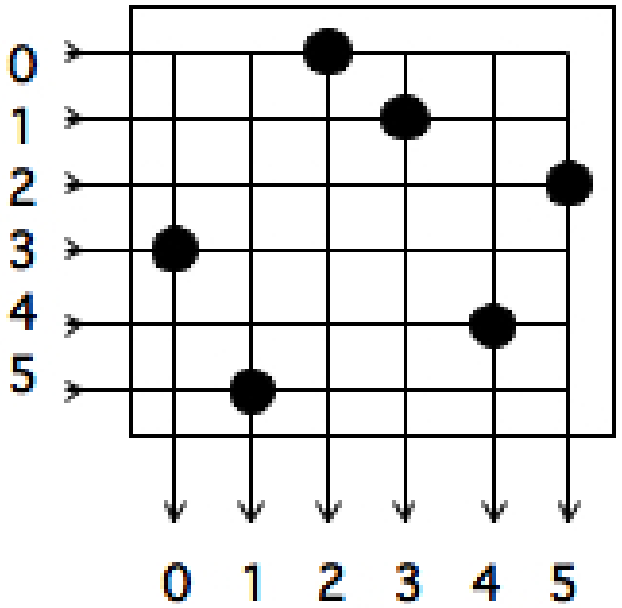
3D Gray code:

2D code and reflection:	0	1	1	0	⋮	0	1	1	0
	0	0	1	1	⋮	1	1	0	0
append 0 and 1 bit:	0	0	0	0	⋮	1	1	1	1

Switching networks

- Solution to all-to-all connection
- (Real all-to-all too expensive)
- Typically layered
- Switching elements: easy to extend

Cross bar

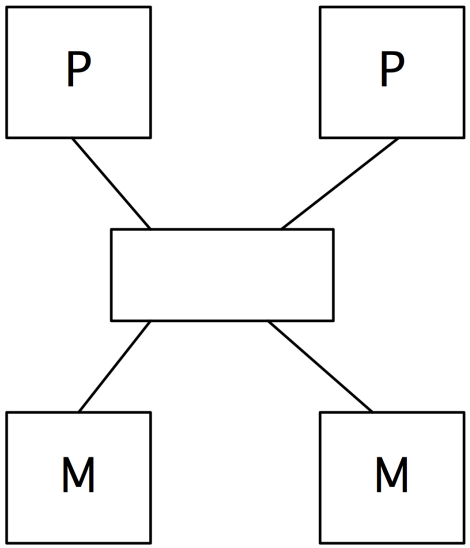


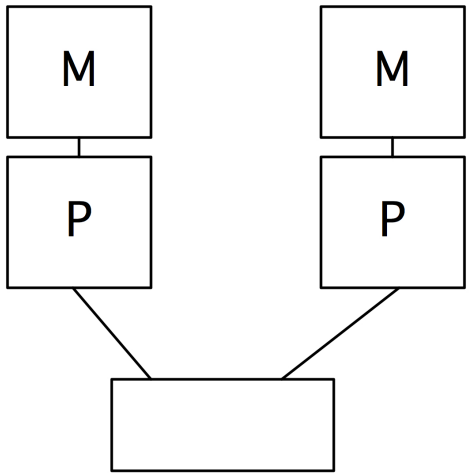
Advantage: non-blocking

Disadvantage: cost

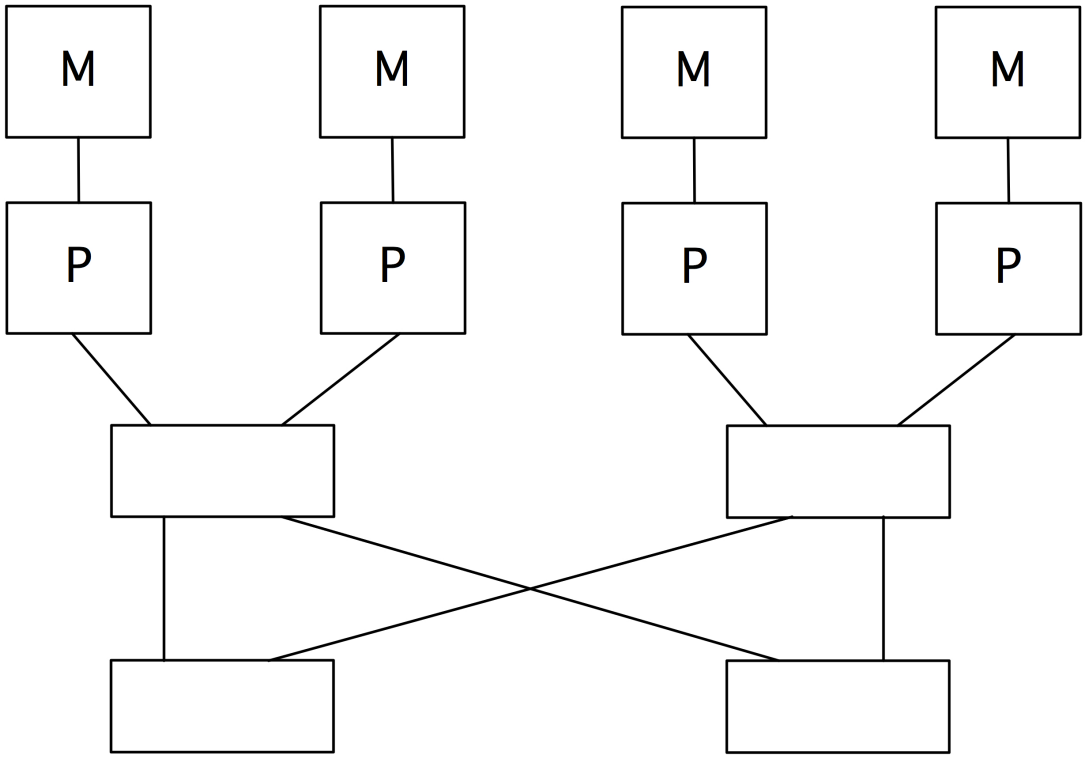
Butterfly exchange

Process to segmented pool of memory, or between processors with private memory:

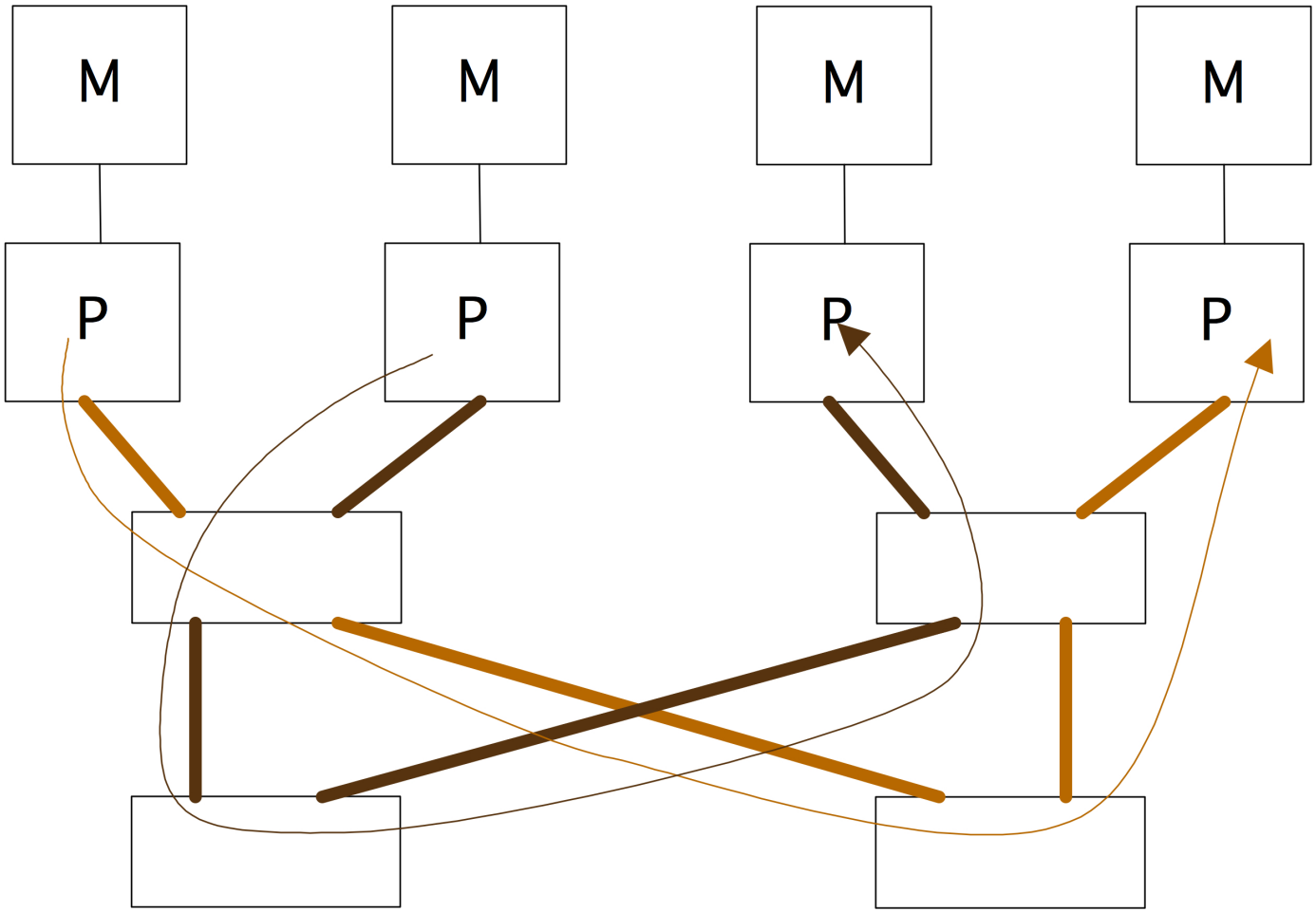




Building up butterflies

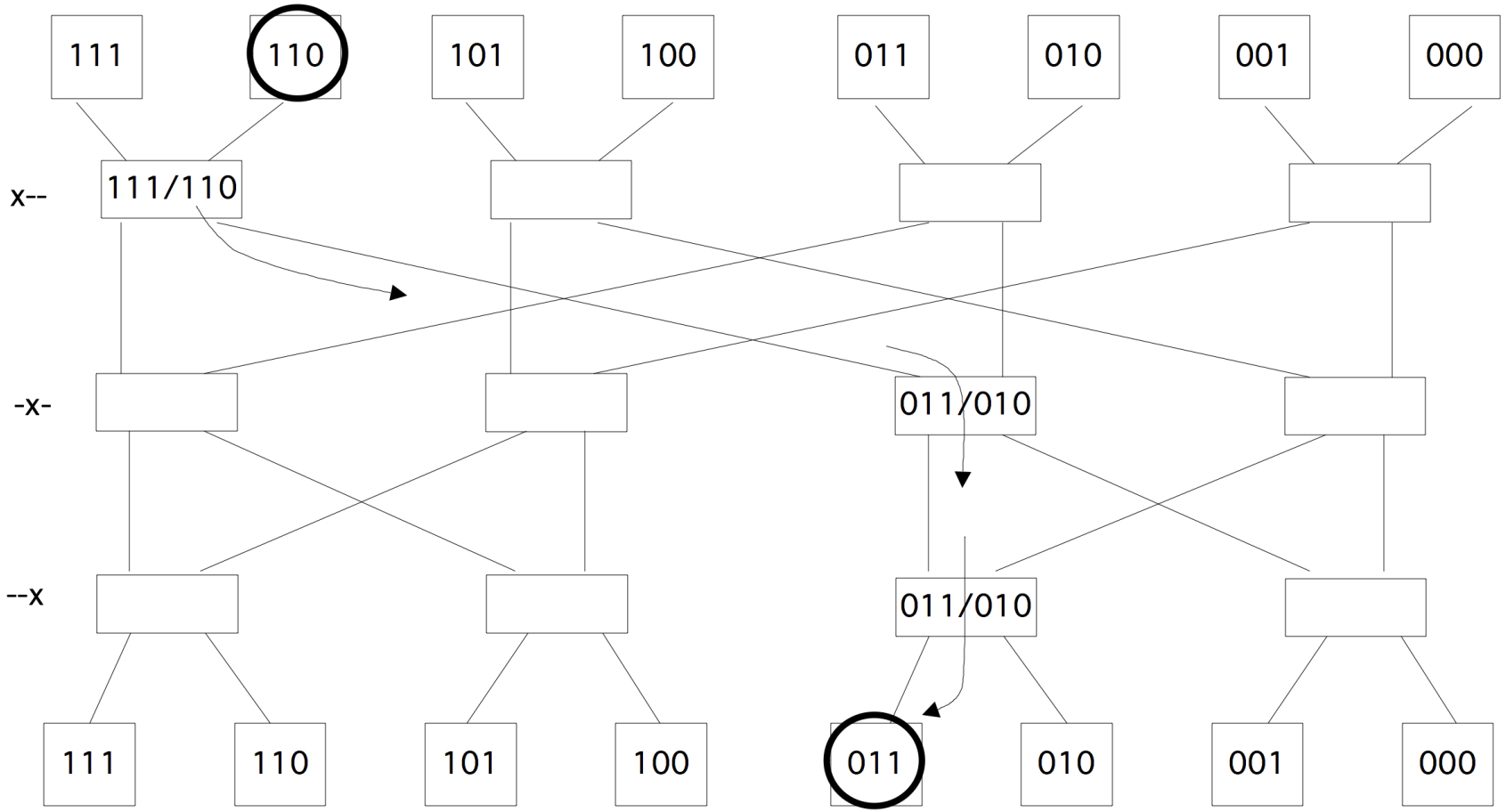


Uniform memory access

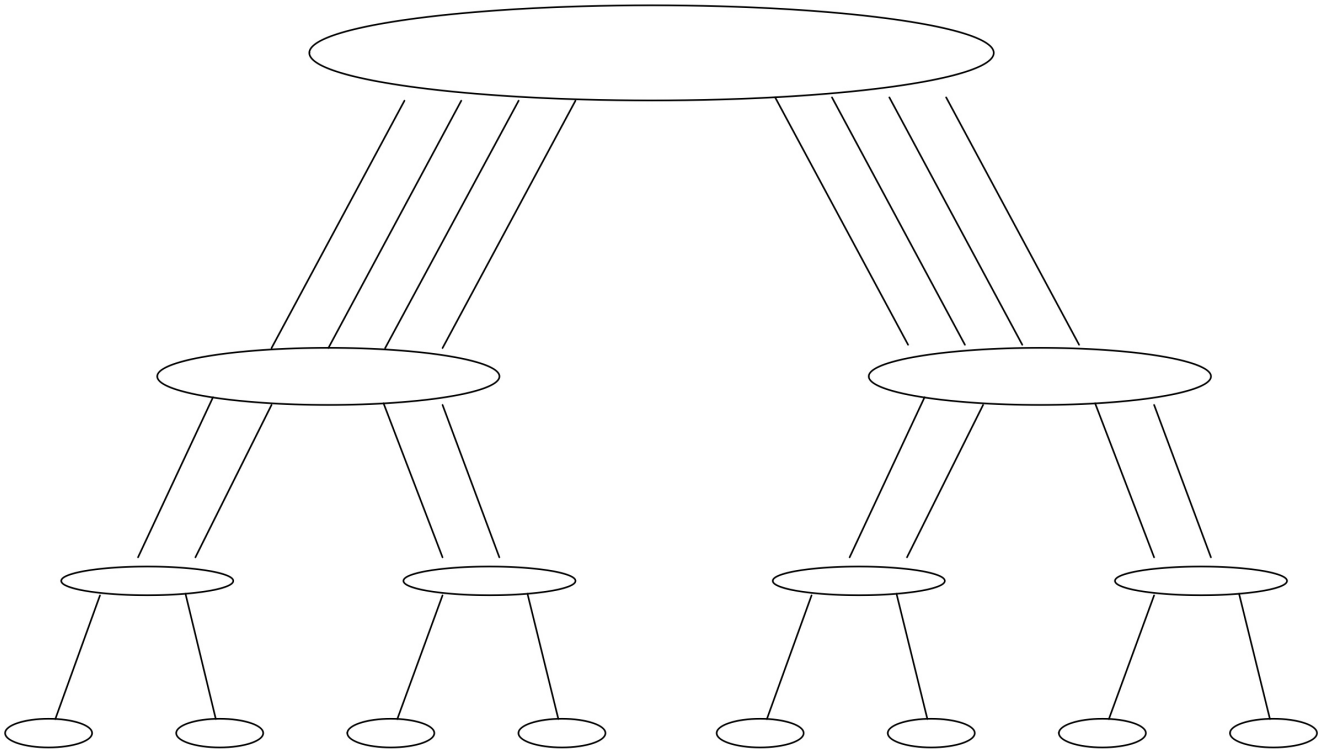


Contention possible

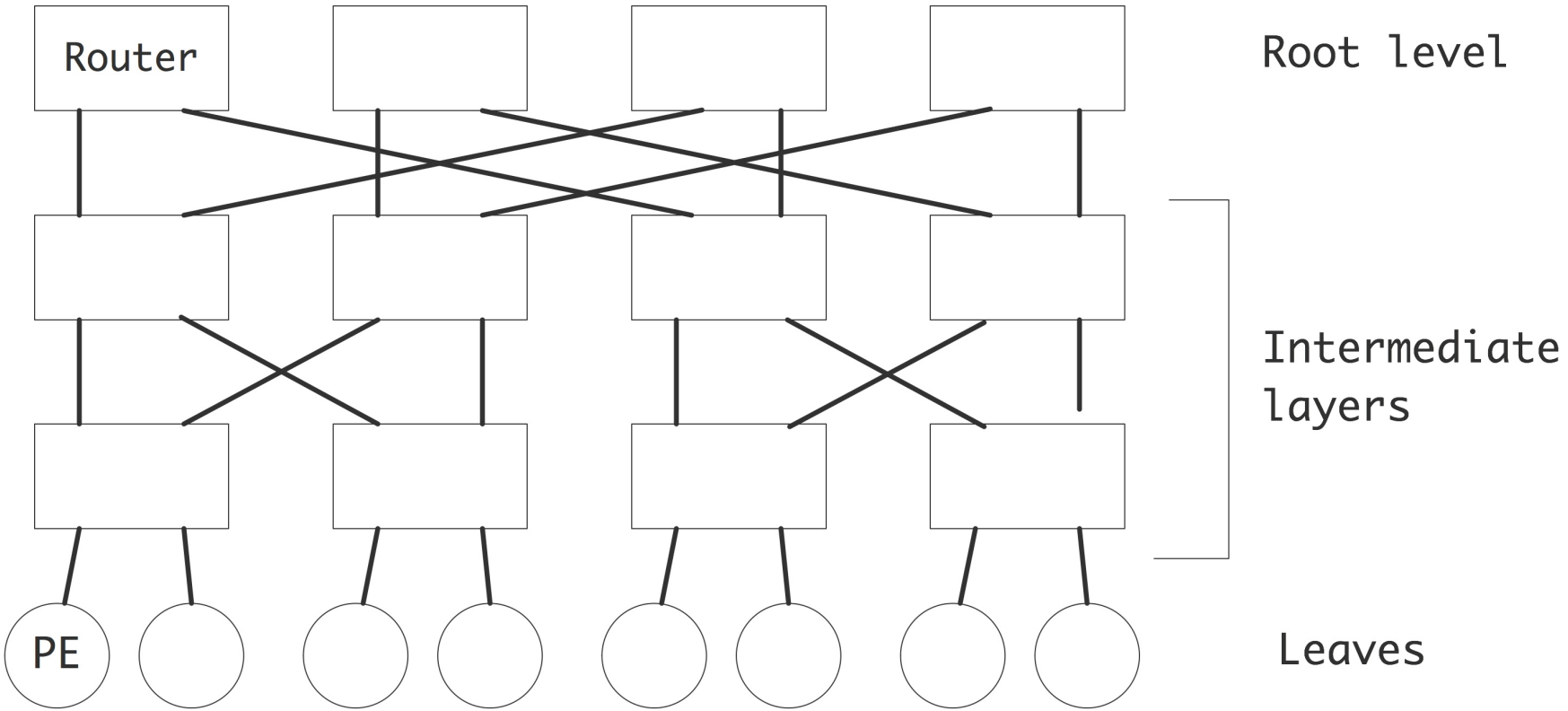
Route calculation



Fat Tree



Fat trees from switching elements



(Clos network)

Fat tree clusters

Mesh clusters

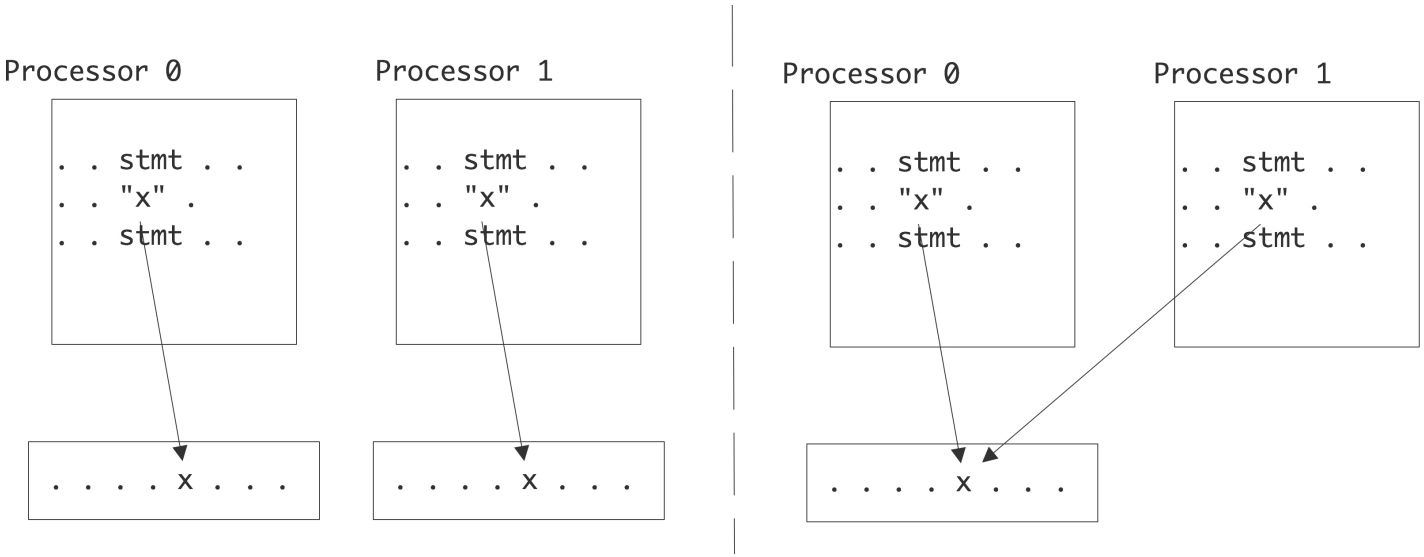
Levels of locality

- Core level: private cache, shared cache
- Node level: numa
- Network: levels in the switch

Table of Contents

Shared vs distributed memory programming

Different memory models:



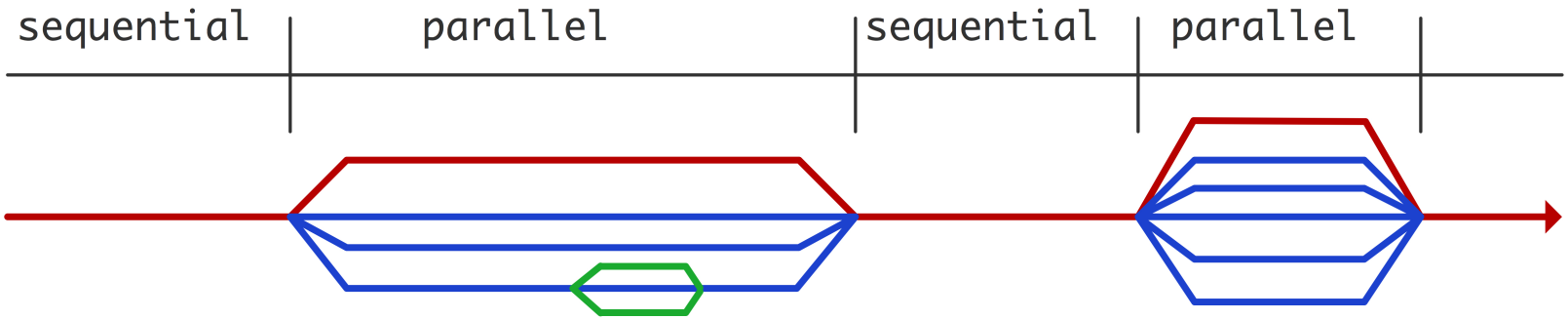
Different questions:

- Shared memory: synchronization problems such as critical sections
- Distributed memory: data motion

Thread parallelism

What is a thread

- Process: code, heap, stack
- Thread: same code but private program counter, stack, local variables
- dynamically (even recursively) created: fork-join



Incremental parallelization!

Thread context

- Private data (stack, local variables) is called 'thread context'
- Context switch: switch from one thread execution to another
- context switches are expensive; alternative hyperthreading
- Intel Xeon Phi: hardware support for 4 threads per core
- GPUs: fast context switching between many threads

Thread programming 1

Pthreads

```
pthread_t threads[NTHREADS];
printf("forking\n");
for (i=0; i<NTHREADS; i++)
    if (pthread_create(threads+i,NULL,&adder,NULL)!=0)
        return i+1;
printf("joining\n");
for (i=0; i<NTHREADS; i++)
    if (pthread_join(threads[i],NULL)!=0)
        return NTHREADS+i+1;
```

Atomic operations

process 1: `I=I+2`
process 2: `I=I+3`

scenario 1.		scenario 2.		scenario 3.	
I = 0					
read I = 0	read I = 0	read I = 0	read I = 0	read I = 0	
do I = 2	do I = 3	do I = 2	do I = 3	do I = 2	
write I = 2			write I = 3	write I = 2	
	write I = 3	write I = 2			read I = 2
					do I = 5
					write I = 5
I = 3		I = 2		I = 5	

Dealing with atomic operations

Semaphores, locks, mutexes, critical sections, transactional memory

Software / hardware

Cilk

Sequential code:

```
int fib(int n){
    if (n<2) return 1;
    else {
         int rst=0;
         rst += fib(n-1);
         rst += fib(n-2);
         return rst;
}
```

Cilk code:

```
cilk int fib(int n){
     if (n<2) return 1;
     else {
          int rst=0;
          rst += spawn fib(n-1);
          rst += spawn fib(n-2);
          sync;
          return rst;
}
```

Sequential consistency: program output identical to sequential

OpenMP

- Directive based
- Parallel sections, parallel loops, tasks

Distributed memory parallelism

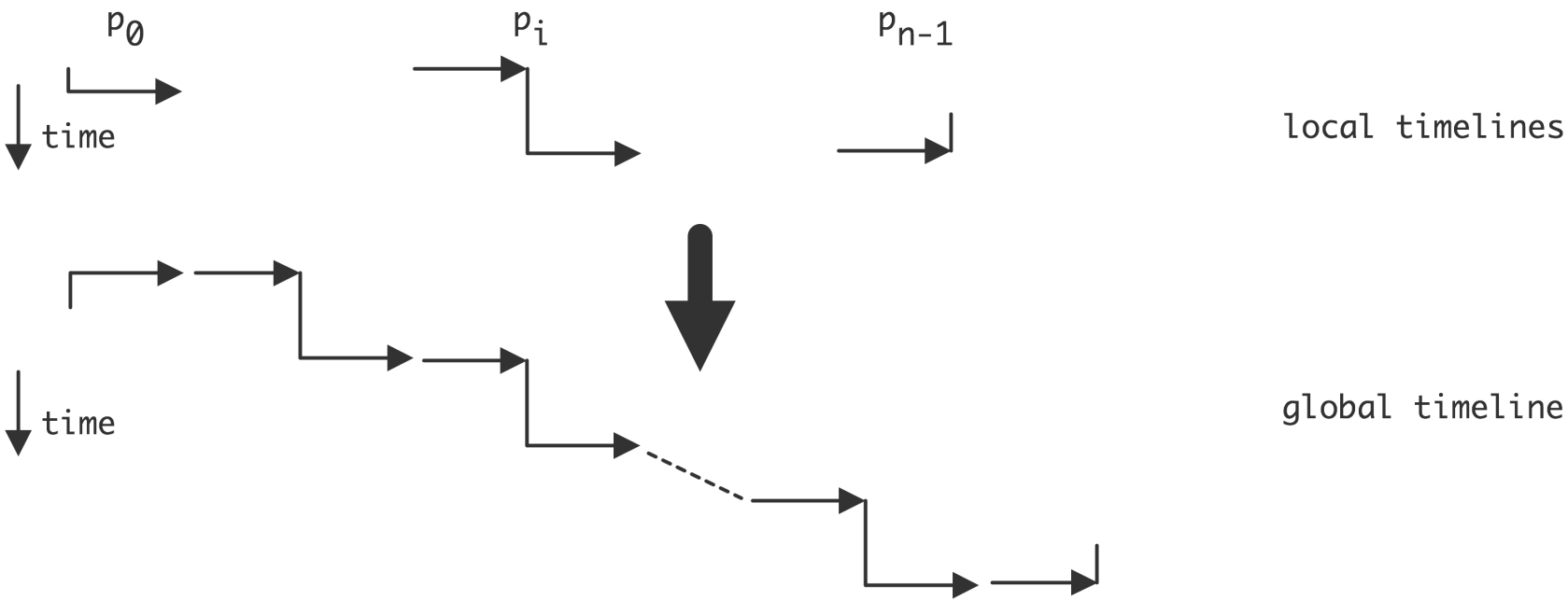
Global vs local view

$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i > 0 \\ y_i \text{ unchanged} & i = 0 \end{cases}$$

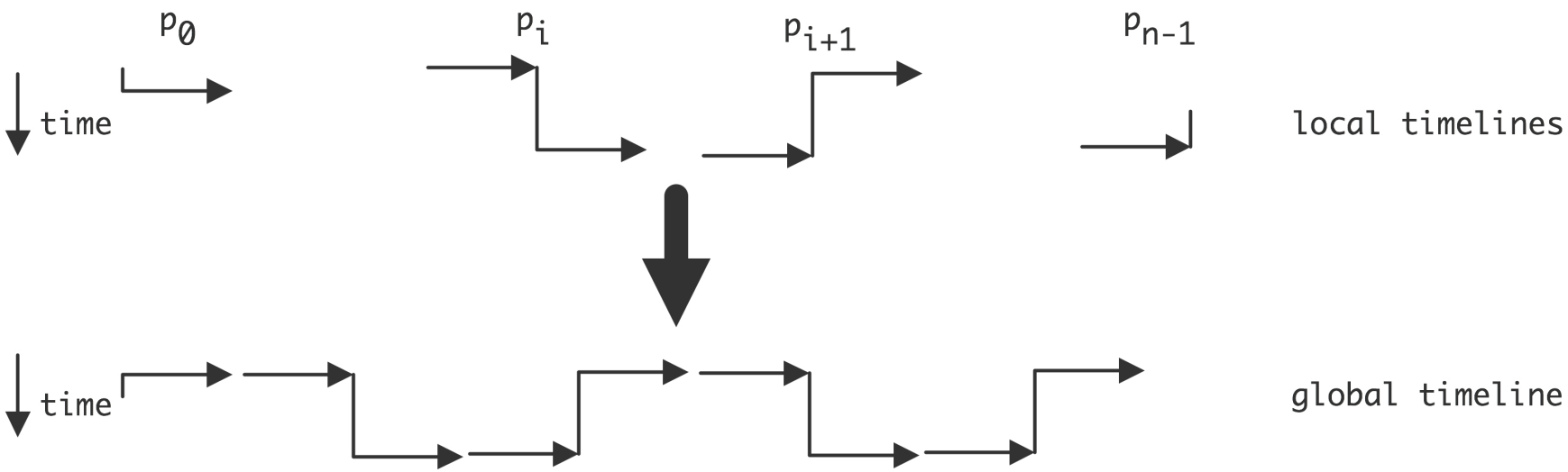
- If I am processor 0 do nothing, otherwise receive a y element from the left, add it to my x element.
- If I am the last processor do nothing, otherwise send my y element to the right.

(Let's think this through...)

Global picture



Careful coding



Better approaches

- Non-blocking send/receive
- One-sided

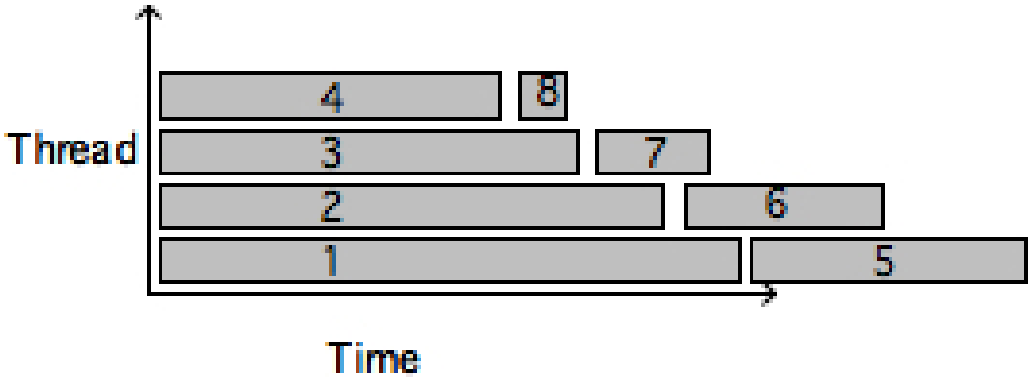
Hybrid/heterogeneous parallelism

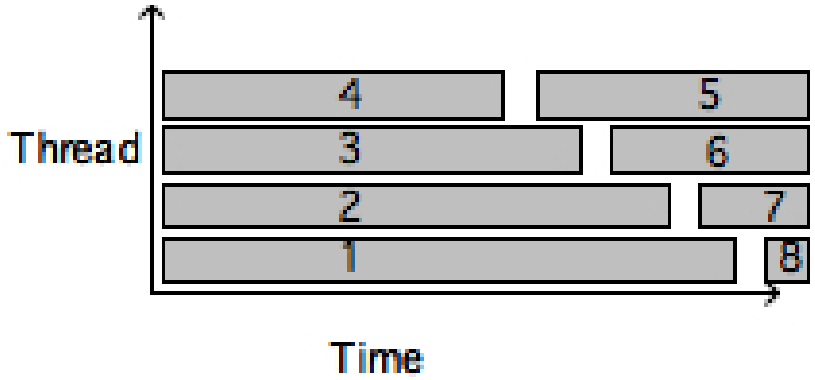
Hybrid computing

- Use MPI between nodes, OpenMP inside nodes
- alternative: ignore shared memory and MPI throughout
- you save: buffers and copying
- bundling communication, load spread

Using threads for load balancing

Dynamic scheduling gives load balancing





Hybrid is possible improvement over strict-MPI

Amdahl's law for hybrid programming

- p nodes with c cores each
- F_p core-parallel fraction, assume full MPI parallel
- ideal speedup pc, running time $T_1/(pc)$, actually:

$$T_{p,c} = T_1 \left(\frac{F_s}{p} + \frac{F_p}{pc} \right) = \frac{T_1}{pc} \left(F_s c + F_p \right) = \frac{T_1}{pc} \left(1 + F_s (c-1) \right).$$

- $T_1/T_{p,c} \approx p/F_s$
- Original Amdahl: $S_p < 1/F_s$, hybrid programming $S_p < p/F_s$

Design patterns

Array of Structures

```
struct { int number; double xcoord,ycoord; } _Node;
struct { double xtrans,ytrans} _Vector;
typedef struct _Node* Node;
typedef struct _Vector* Vector;

Node *nodes = (node) malloc( n_nodes*sizeof(struct _Node) );
```

Operations

Operate

```
void shift(node the_point,vector by) {
  the_point->xcoord += by->xtrans;
  the_point->ycoord += by->ytrans;
}
```

in a loop

```
for (i=0; i<n_nodes; i++) {
  shift(nodes[i],shift_vector);
}
```

Along come the 80s

Vector operations

```
node_numbers = (int*) malloc( n_nodes*sizeof(int) );
node_xcoords = // et cetera
node_ycoords = // et cetera
```

and you would iterate

```
for (i=0; i<n_nodes; i++) {
  node_xoords[i] += shift_vector->xtrans;
  node_yoords[i] += shift_vector->ytrans;
}
```

and the wheel of reinvention turns further

The original design was better for MPI in the 1990s

except when vector instructions (and GPUs) came along in the 2000s

Latency hiding

- Memory and network are slow, prevent having to wait for it
- Hardware magic: out-of-order execution, caches, prefetching

Explicit latency hiding

Matrix vector product

$$\forall_{i \in I_p} : y_i = \sum_j a_{ij} x_j.$$

x needs to be gathered:

$$\forall_{i \in I_p} : y_i = \left(\sum_{j \text{ local}} + \sum_{j \text{ not local}} \right) a_{ij} x_j.$$

Overlap loads and local operations

Possible in MPI and Xeon Phi offloading,
very hard to do with caches

What's left

Parallel languages

- Co-array Fortran: extensions to the Fortran standard
- X10
- Chapel
- UPC
- BSP
- MapReduce
- Pregel, ...

UPC example

```
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main()
{
  int i;
  upc_forall(i=0; i<N; i++; i)
    v1plusv2[i]=v1[i]+v2[i];
}
```

Co-array Fortran example

Explicit dimension for 'images':

```
Real,dimension(100),codimension[*] :: X
Real :: X(100)[*]
Real :: X(100,200)[10,0:9,*]
```

determined by runtime environment

Grab bag of other approaches

- OS-based: data movement induced by cache misses
- Active messages: application level Remote Procedure Call (see: Charm++)

Table of Contents

The load balancing problem

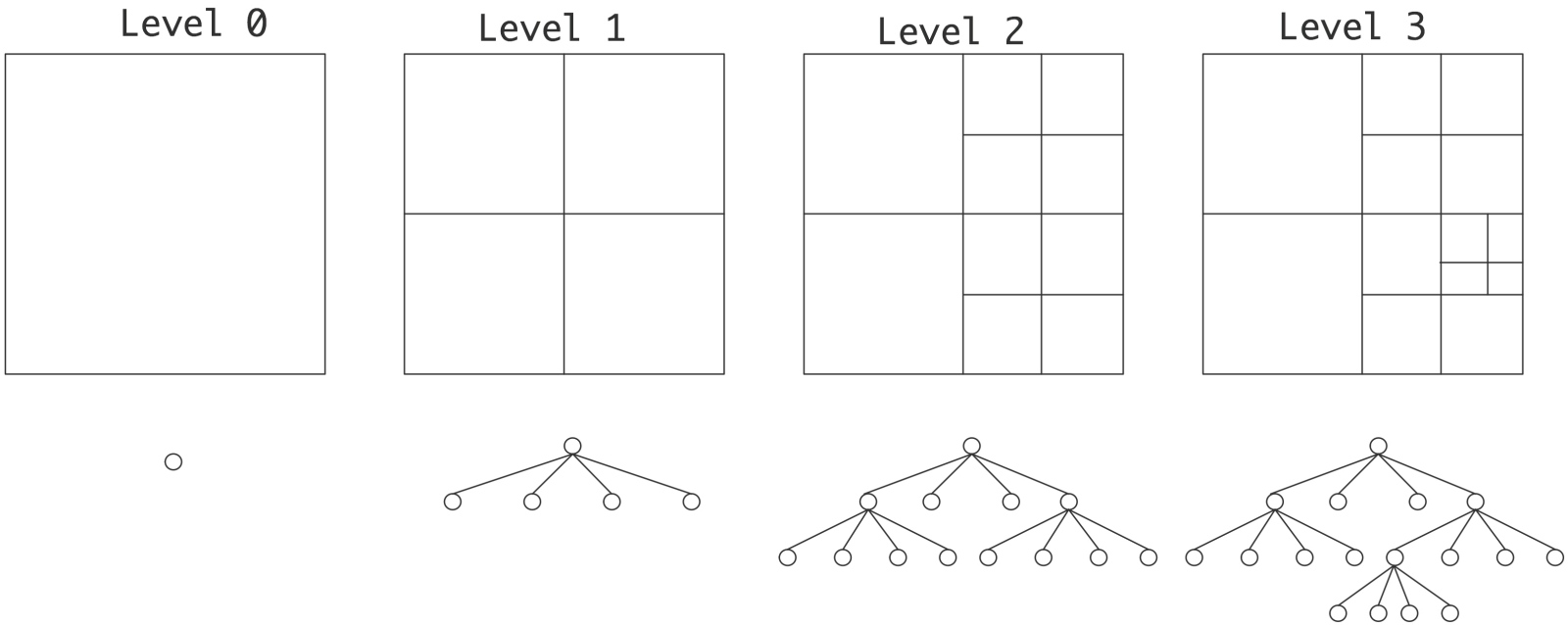
- Application load can change dynamically
  e.g., mesh refinement, time-dependent problems
- Splitting off and merging loads
- No real software support: write application anticipating load management
- Initial balancing: graph partitioners

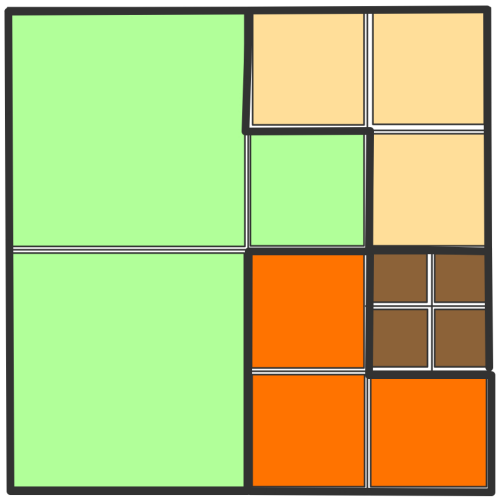
Load balancing and performance

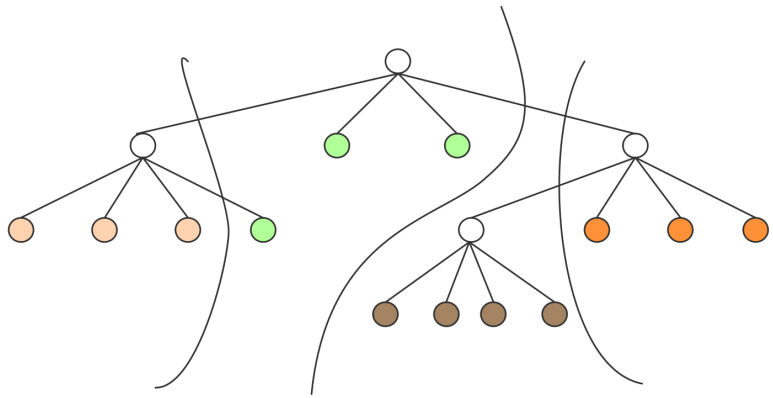
- Assignment to arbitrary processor violates locality
- Need a dynamic load assignment scheme that preserves locality under load migration
- Fairly easy for regular problems, for irregular?

Space-filling curves

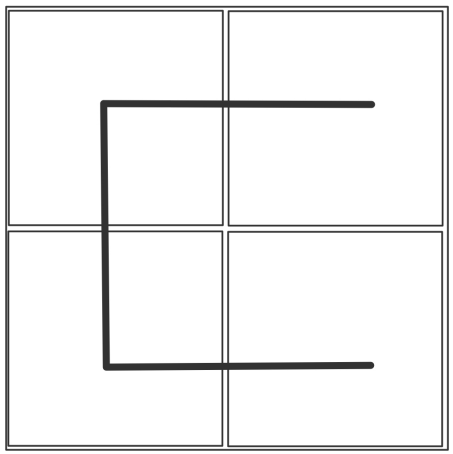
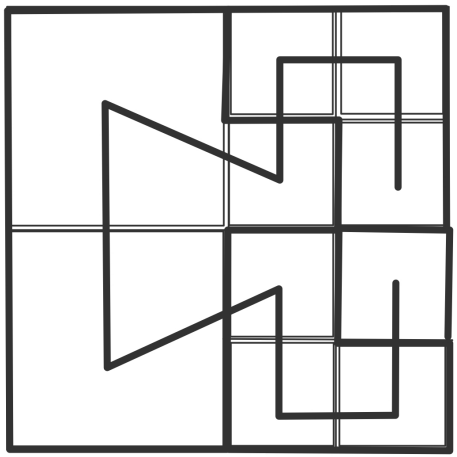
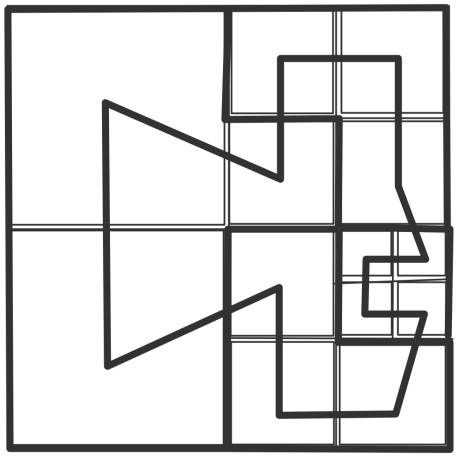
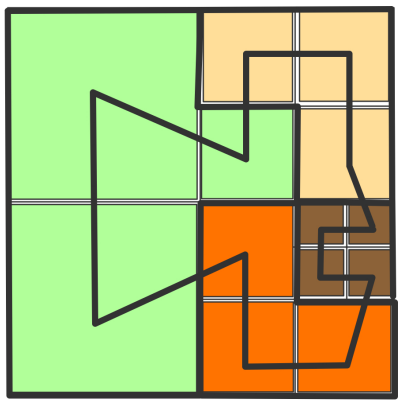
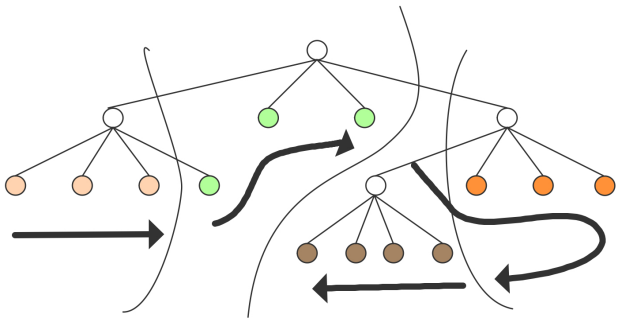
Adaptive refinement and load assignment





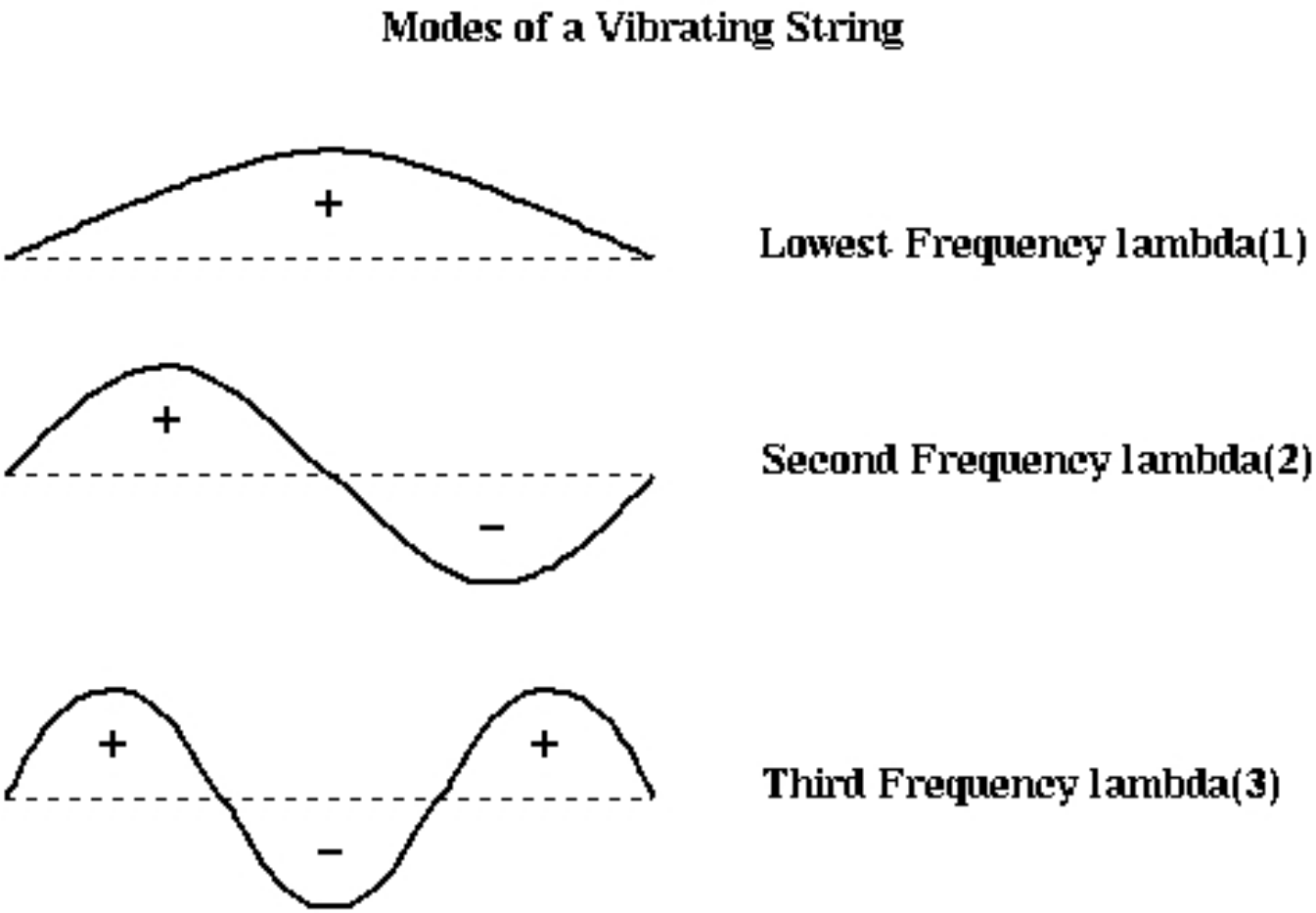


Assignment through Space-Filling Curve

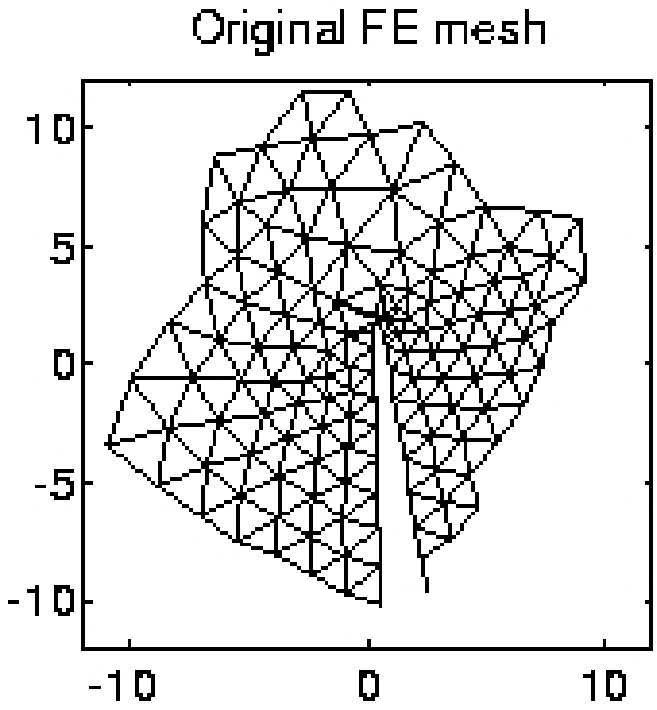
Domain partitioning by Fiedler vectors

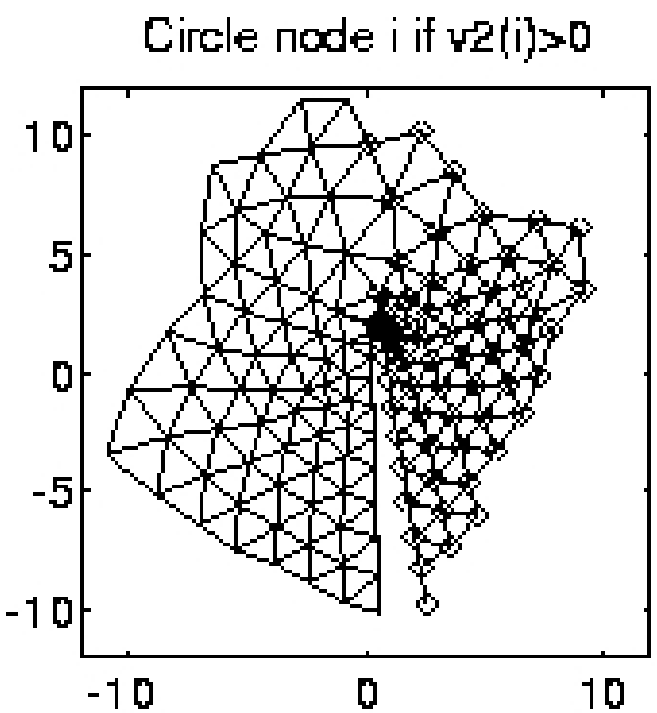
Inspiration from physics

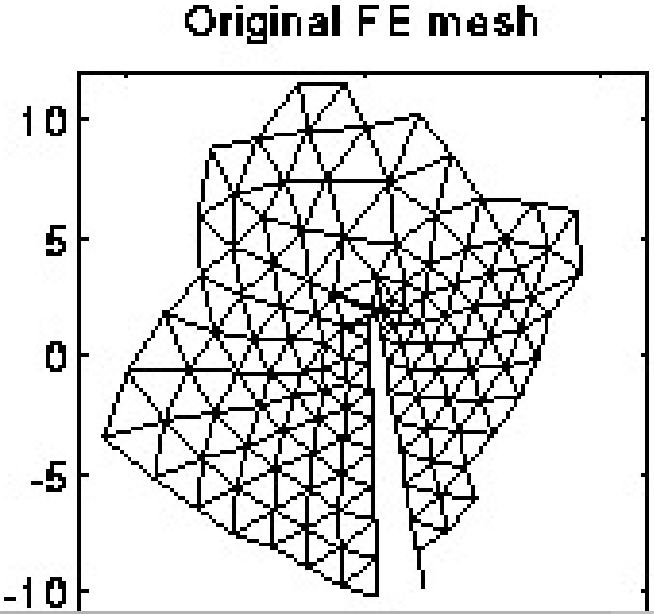


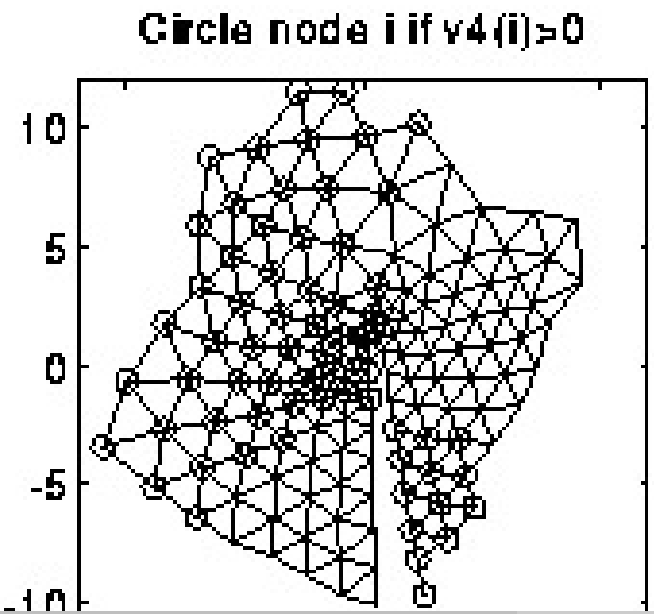
Graph laplacian

- Set $G_{ij} = -1$ if edge (i, j)
- Set G_{ii} positive to give zero rowsums
- First eigenvector is zero, positive eigenvector
- Second eigenvector has pos/neg, divides in two
- n-th eigenvector divides in n parts


Original FE mesh
10
5
0
-5
-10
-10
0
10


Circle node i if v2(i)>0
10
5
0
-5
-10
-10
0
10


Original FE mesh
10
5
0
-5
-10


Circle node i if v4(i)>0
10
5
0
-5