What is parallelism?

Victor Eijkhout

Fall 2022

Justification

Parallel computing has been a necessity for decades in computational science. Here we discuss some of the basic concepts. Actual parallel programming will be discussed in other lectures.

Basic concepts

1 The basic idea

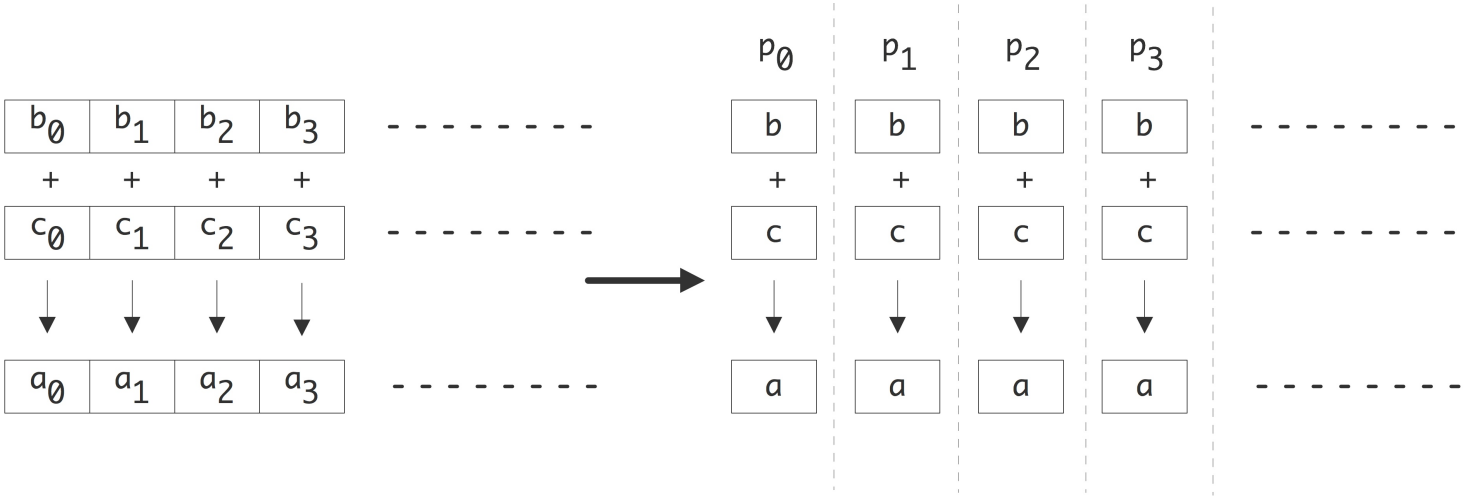
Parallelism is about doing multiple things at once.

- Hardware: vector instructions, multiple cores, nodes in a cluster.
- Algorithm: can you think of examples?

2 Simple example

Summing two arrays together:

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



Parallel: every processing element does

```
for ( i in my_subset_of_indices )
  a[i] = b[i] + c[i];
```

~~Time goes down linearly with processors~~

3 Differences between operations

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

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

- Compare operation counts
- Compare behavior on single processor. What about multi-core?
- Other thoughts about parallel execution?

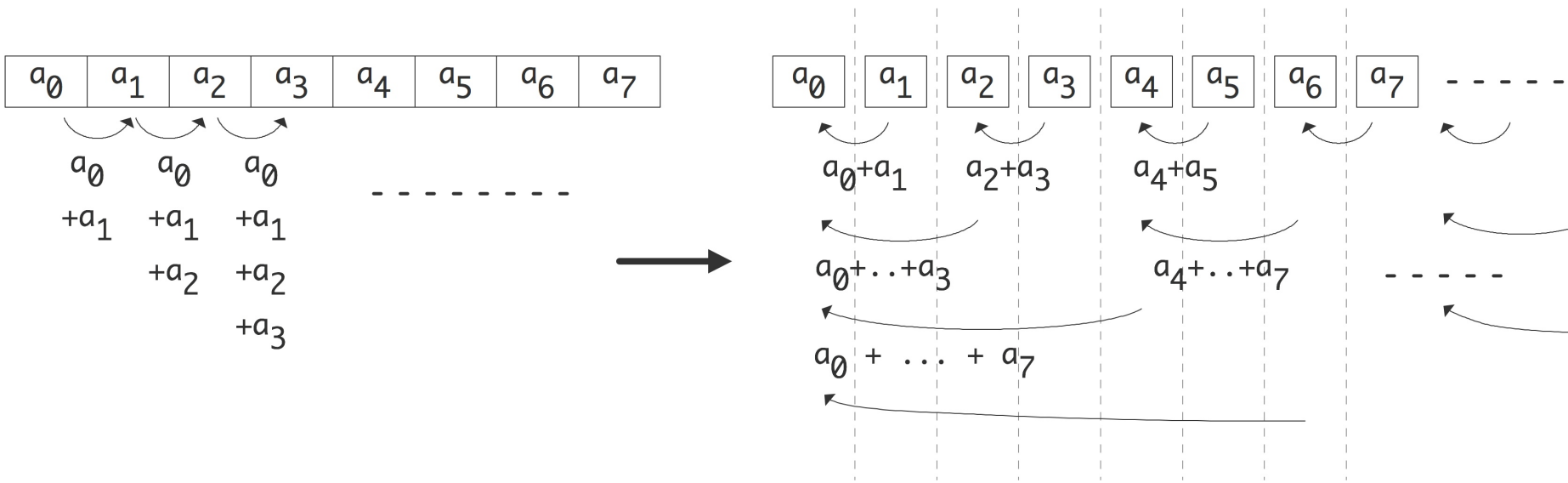
4 Summing

Naive algorithm

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

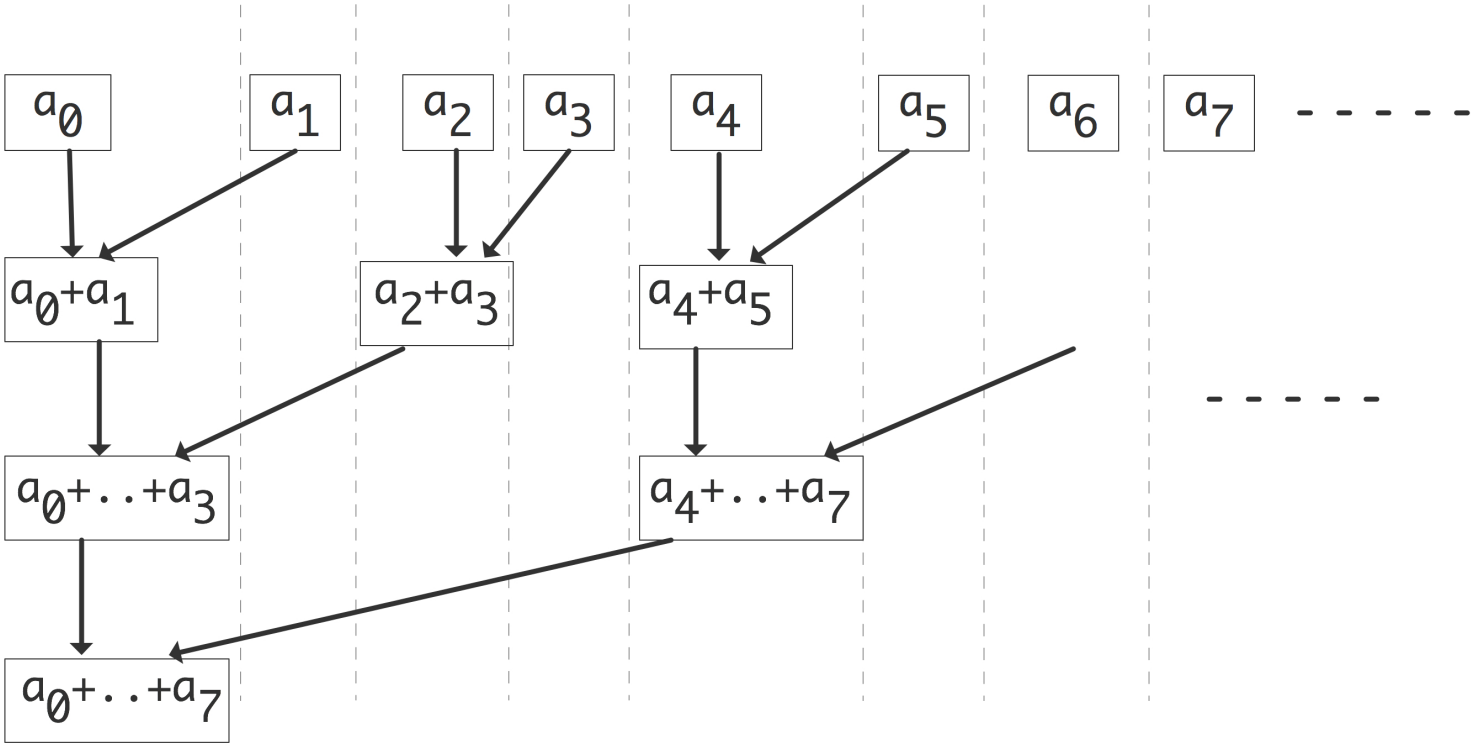
Recoding

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



5 And then there is hardware

Topology of the processors:



increasing distance: limit on parallel speedup

Theoretical concepts

Efficiency and scaling

6 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 $E_p = S_p/p$, $0 < E_p \leq 1$

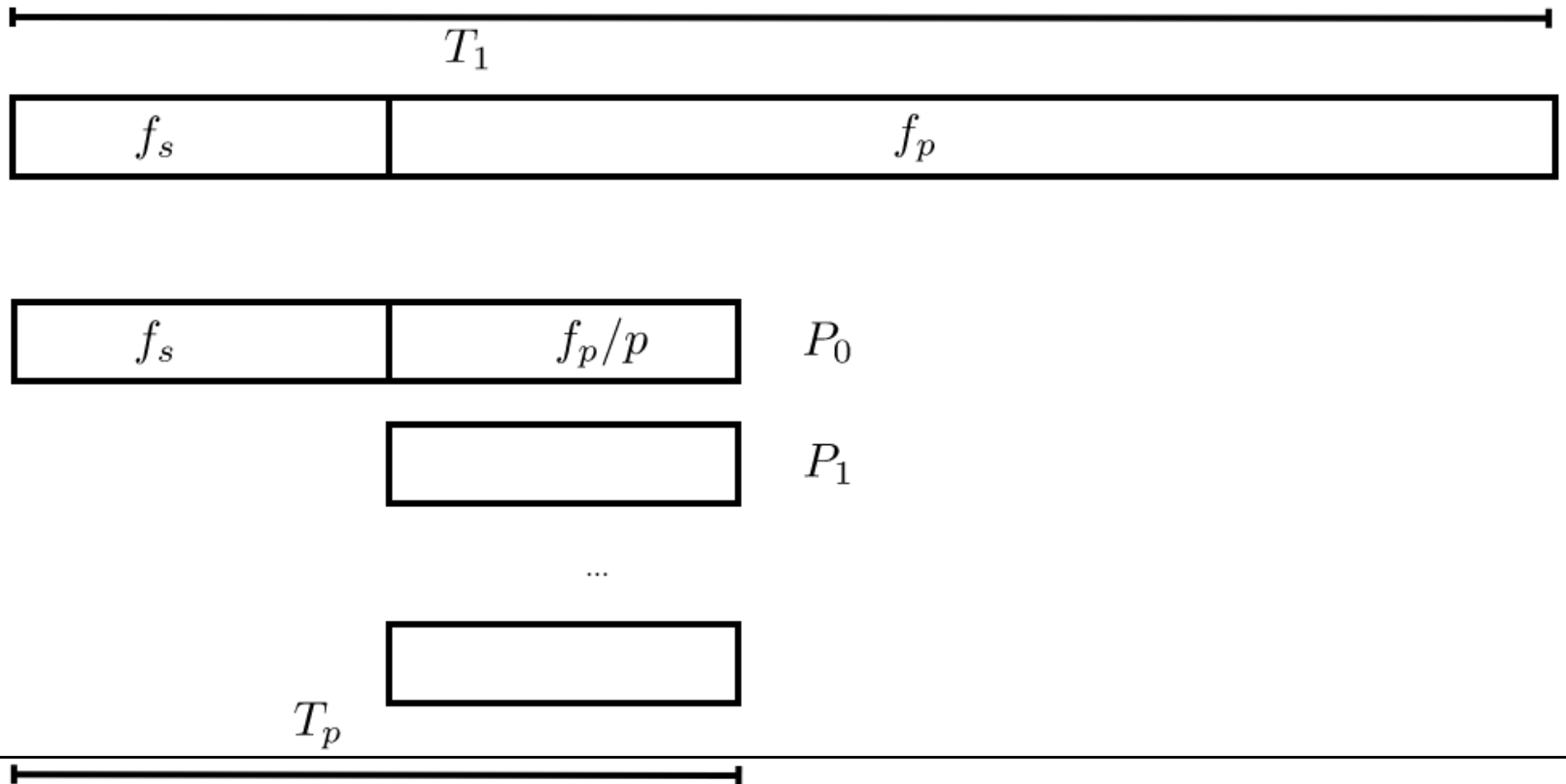
But:

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

7 Amdahl's law

Let's assume that part of the application can be parallelized, part not. (Examples?)

- F_s sequential fraction, F_p parallelizable fraction
- $F_s + F_p = 1$



8 Amdahl's law, analysis

- 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 $E \sim 1/P$.

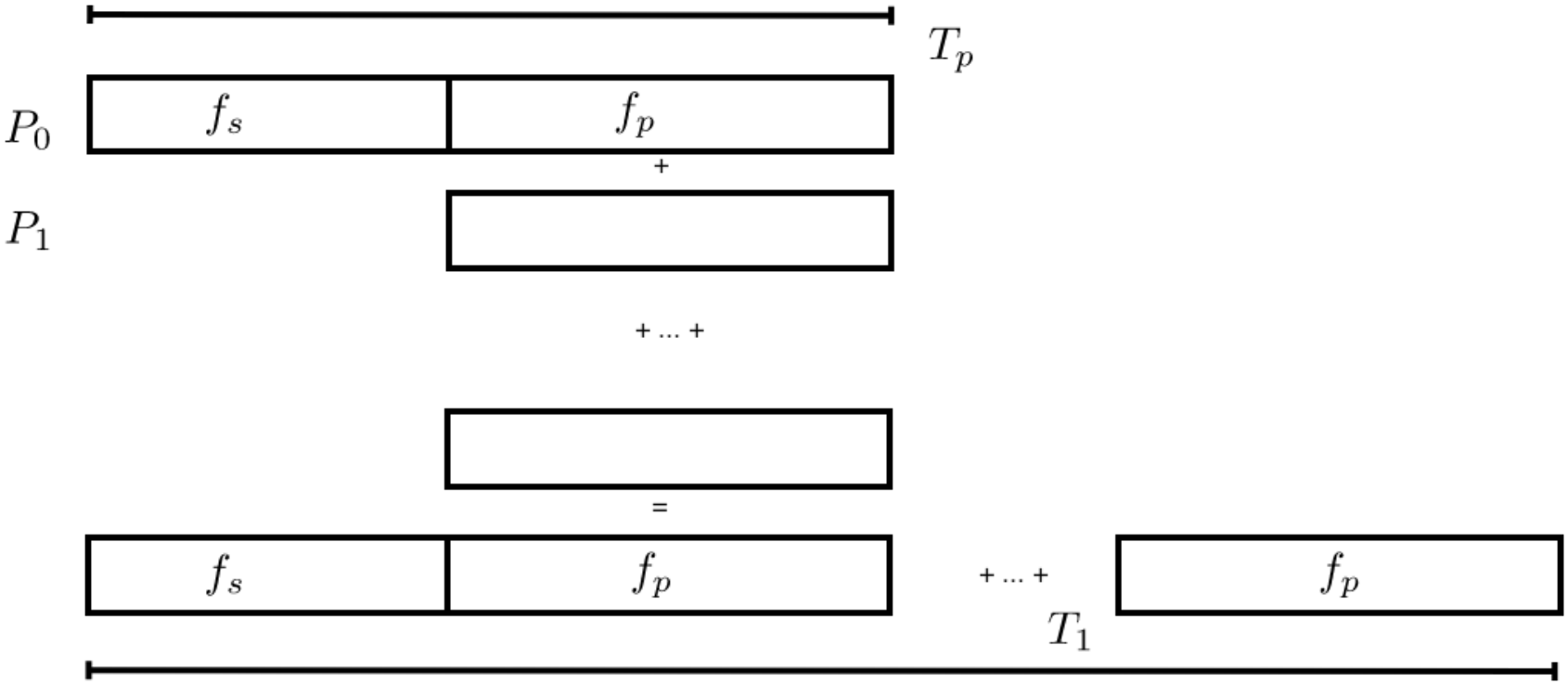
Do you see problems with this?

9 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

10 Gustafson's law

Reconstruct the sequential execution from the parallel, then analyze efficiency.



11 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

12 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

13 Strong scaling

- Let M be the total memory needed for your problem.
- Let P be the number of processors
  $\Rightarrow$ memory per processor is M/P
- What is $\lim_{P\to\infty} E_P$?
  (Note that implicitly $E_p = E(P, M)$.)

14 Weak scaling

- Let M be the memory per processor.
- Let P be the number of processors
  $\Rightarrow$ total memory is $M \cdot P$
- What is $\lim_{P \to \infty} E_P$?
  (Note that implicitly $E_p = E(P, M)$.)

Exercise 1: Linpack scaling

Explore simulation scaling in the context of the *Linpack benchmark*, that is, Gaussian elimination. Ignore the system solving part and only consider the factorization part; assume that it can be perfectly parallelized.

1. Suppose you have a single core machine, and your benchmark run takes time T with M words of memory. Now you buy a processor twice as fast, and you want to do a benchmark run that again takes time T. How much memory do you need?
2. Now suppose you have a machine with P processors, each with M memory, and your benchmark run takes time T. You buy a machine with $2P$ processors, of the same clock speed and core count, and you want to do a benchmark run, again taking time T. How much memory does each node take?

15 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:

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

16 Simulation scaling con'td

- 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.

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

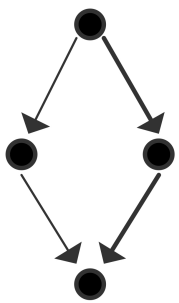
- Substituting $M = Pm$, we find ultimately

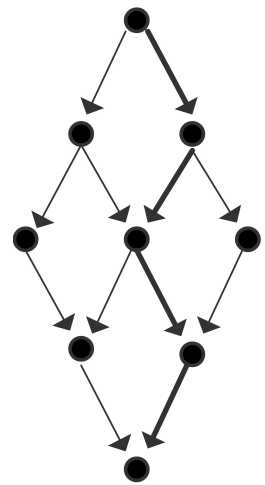
$$m = C \begin{cases} 1 \,/\, P^{1/(d+1)} & \text{hyperbolic} \\ 1 \,/\, P^{2/(d+2)} & \text{parabolic} \end{cases}$$

Critical path analysis

17 Critical path

- The sequential fraction contains a *critical path*: a sequence of operations that depend on each other.
- Example?
- $T_\infty =$ time with unlimited processors: length of critical path.





18 Brent's theorem

Let m be the total number of tasks, p the number of processors, and t the length of a *critical path*. Then the computation can be done in

$$T_p \leq t + \frac{m-t}{p}.$$

- Time equals the length of the critical path . . .
- . . . plus the remaining work as parallel as possible.

Exercise 2: Linpack analysis

Apply Brent's theorem to Gaussian elimination, assuming that add/multiply/division all take one unit time.

What is the critical path; what is its length; what is the resulting upper bound on the parallel runtime?

How many processors could you theoretically use? What speedup and efficiency does that give?

Granularity

19 Definition

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

20 Instruction level parallelism

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

For the compiler / processor to worry about

21 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

22 Task-level parallelism

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

23 Conveniently parallel

Example: Mandelbrot set

Parameter sweep,
often best handled by external tools

24 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
```

The SIMD/MIMD/SPMD/SIMT model for parallelism

25 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

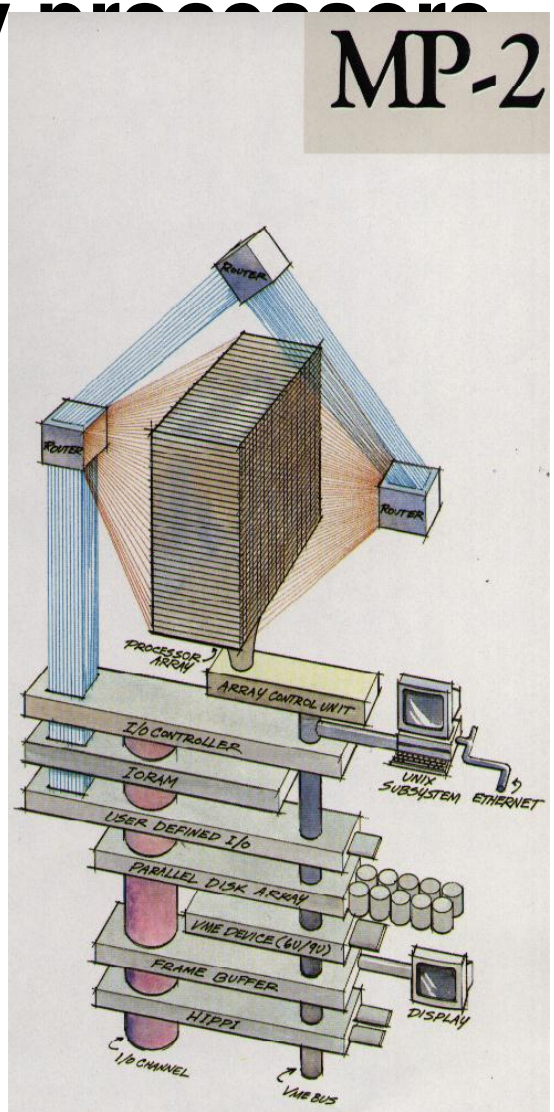
26 SIMD

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

27 SIMD: array processors

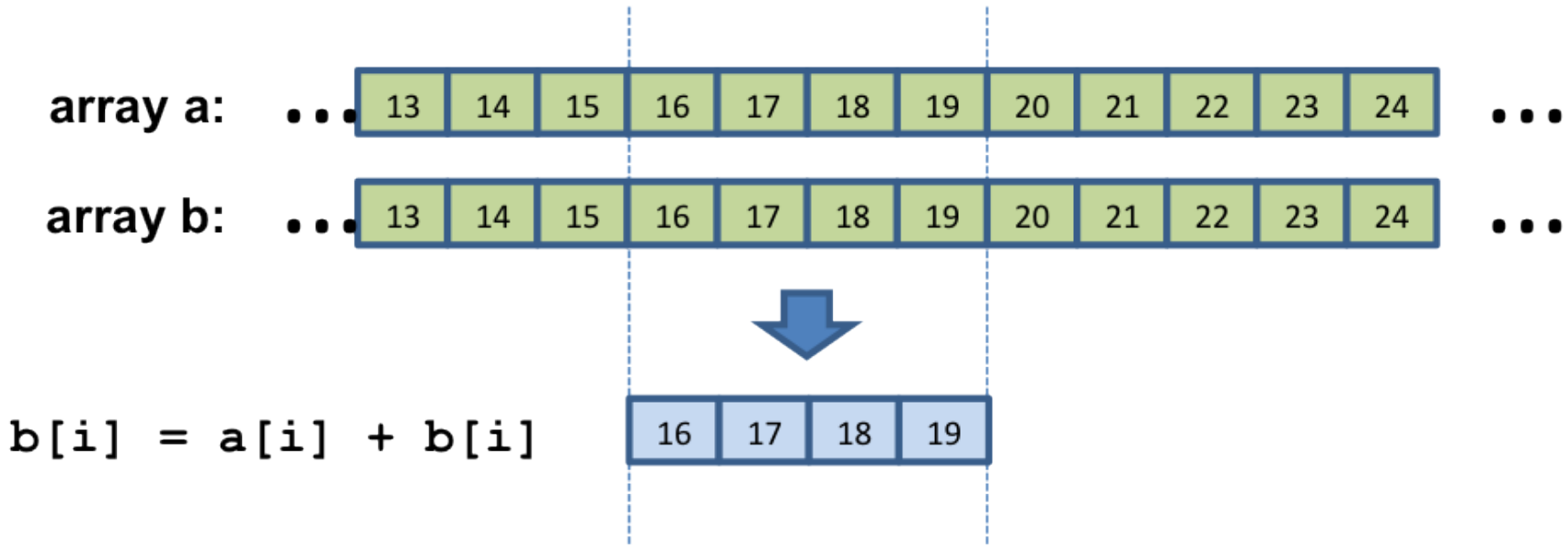
Technology going back to the 1980s: FPS, MasPar, CM, GoodYear

Major advantage: simplification of processor



28 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



29 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`)

30 New branches in the taxonomy

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

31 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)

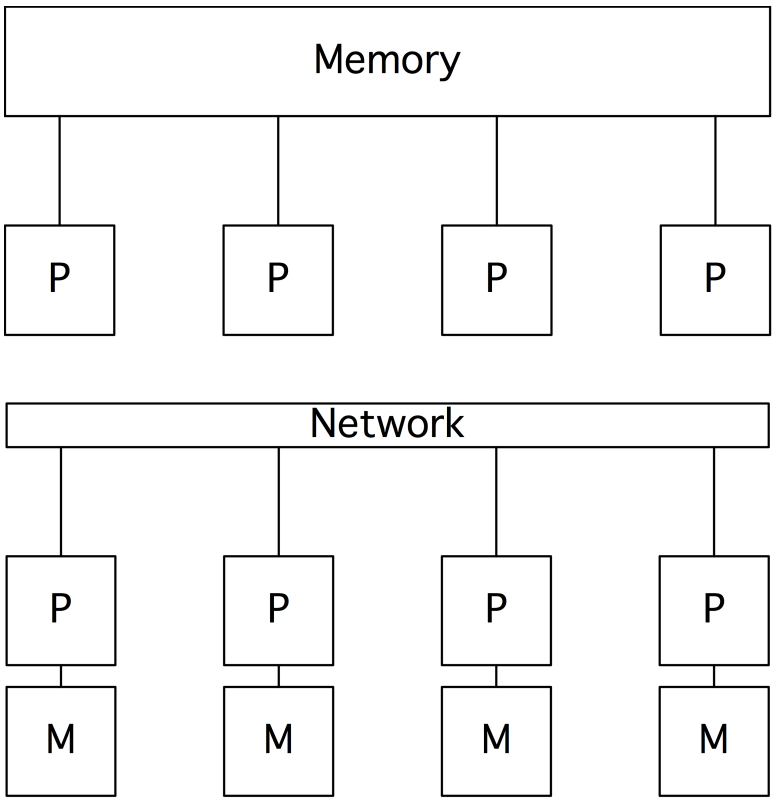
32 GPUs and data paralleism

Lockstep in thread block,
single instruction model between streaming processors

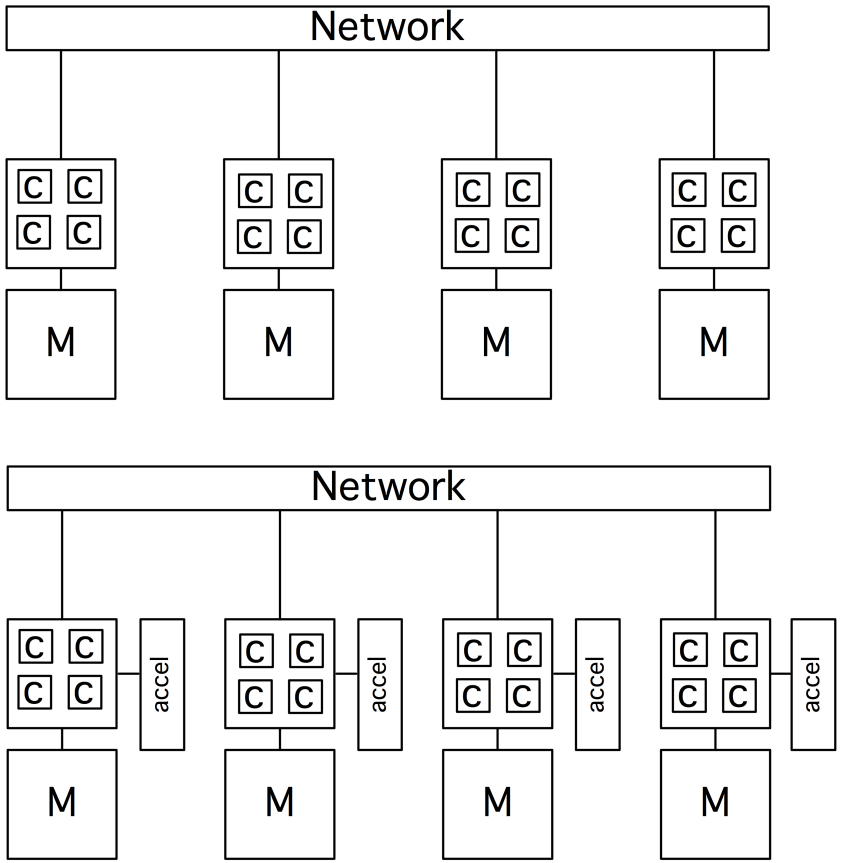
(more about GPU threads later)

Characterization of parallelism by memory model

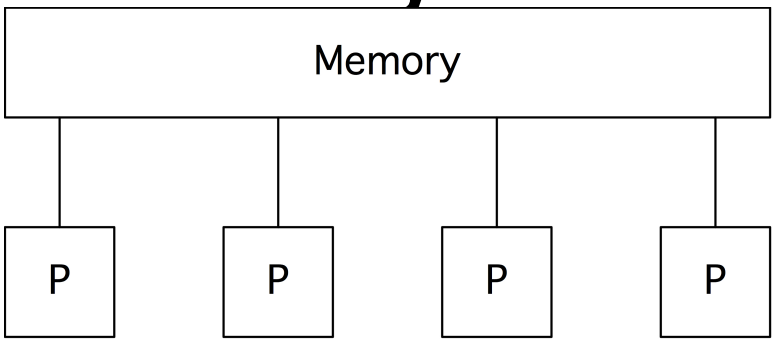
33 Major types of memory organization, classic



34 Major types of memory organization, contemporary



35 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

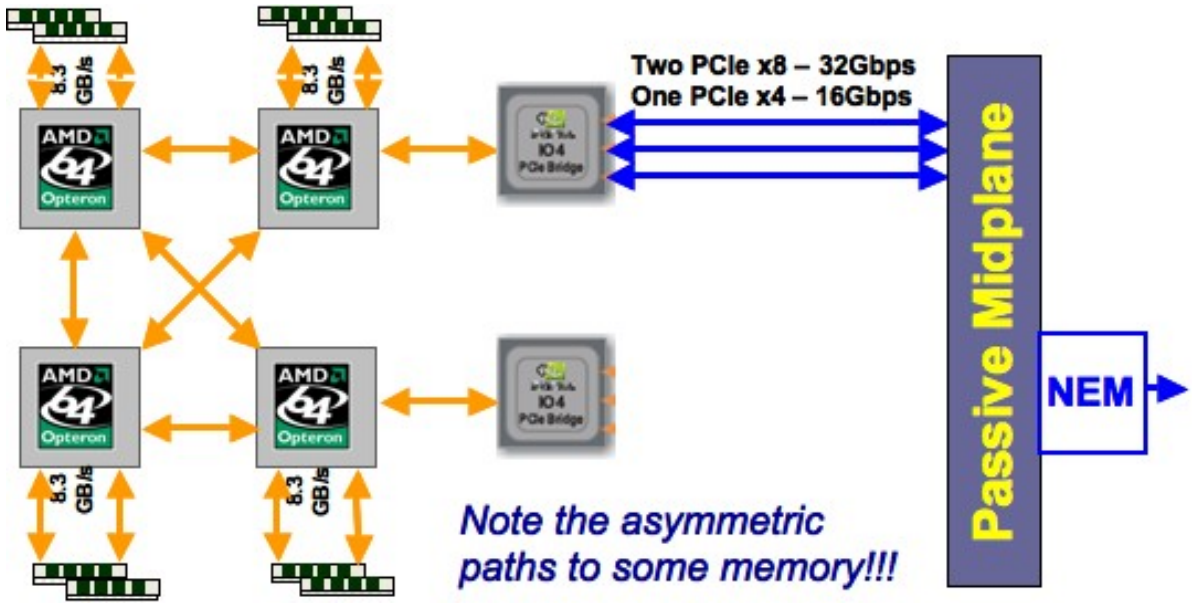
36 SMP, bus design

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

37 Non-uniform Memory Access

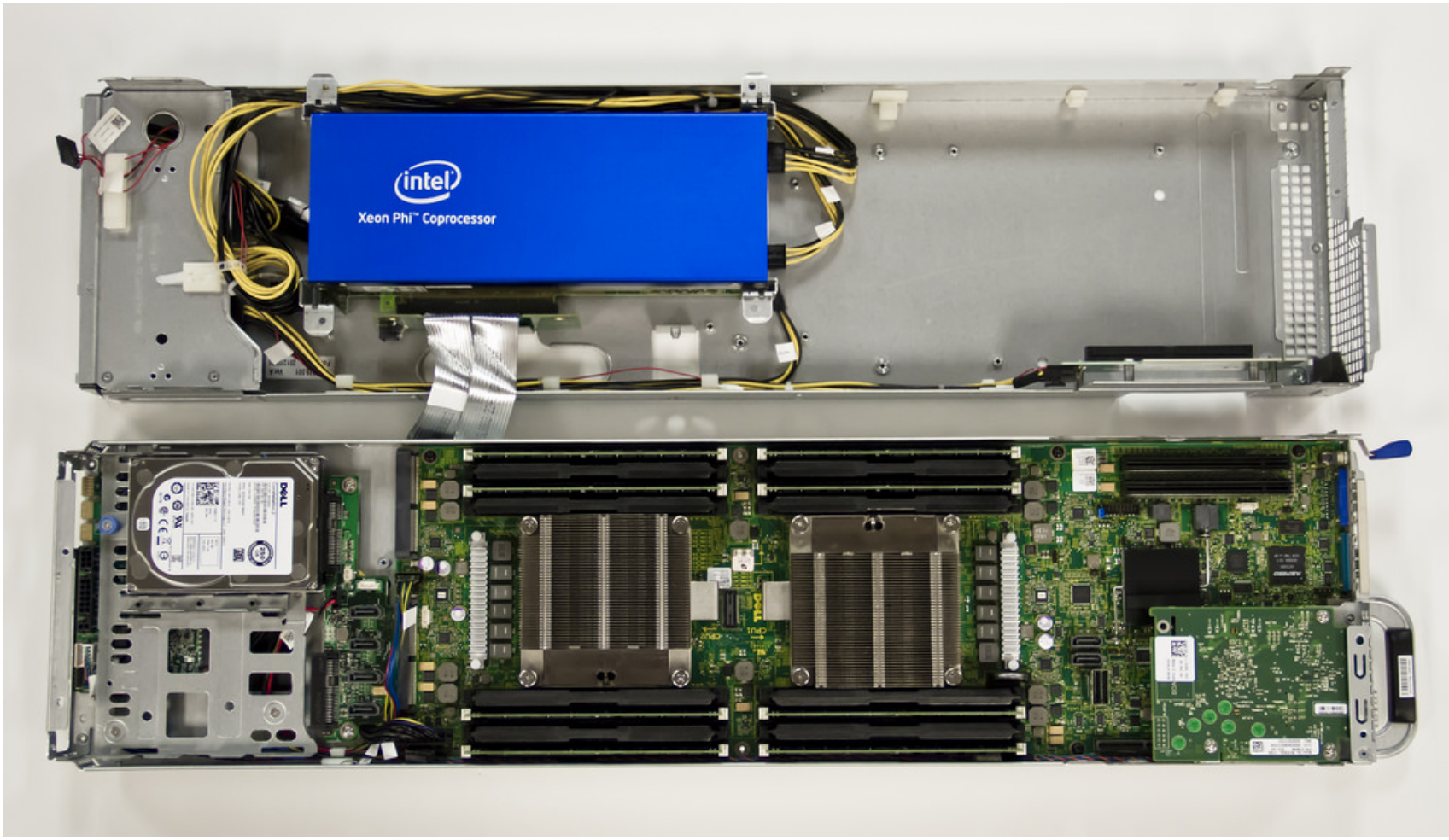
Memory is equally programmable, but not equally accessible

- Different caches, different affinity



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

38 Picture of NUMA



Interconnects and topologies, theoretical concepts

39 Topology concepts

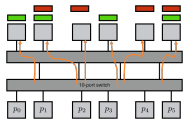
- Hardware characteristics
- Software requirement
- Design: how ‘close’ are processors?

40 Graph theory

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

41 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

42 Design 1: bus

Already discussed; simple design, does not scale very far

43 Design 2: linear arrays

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

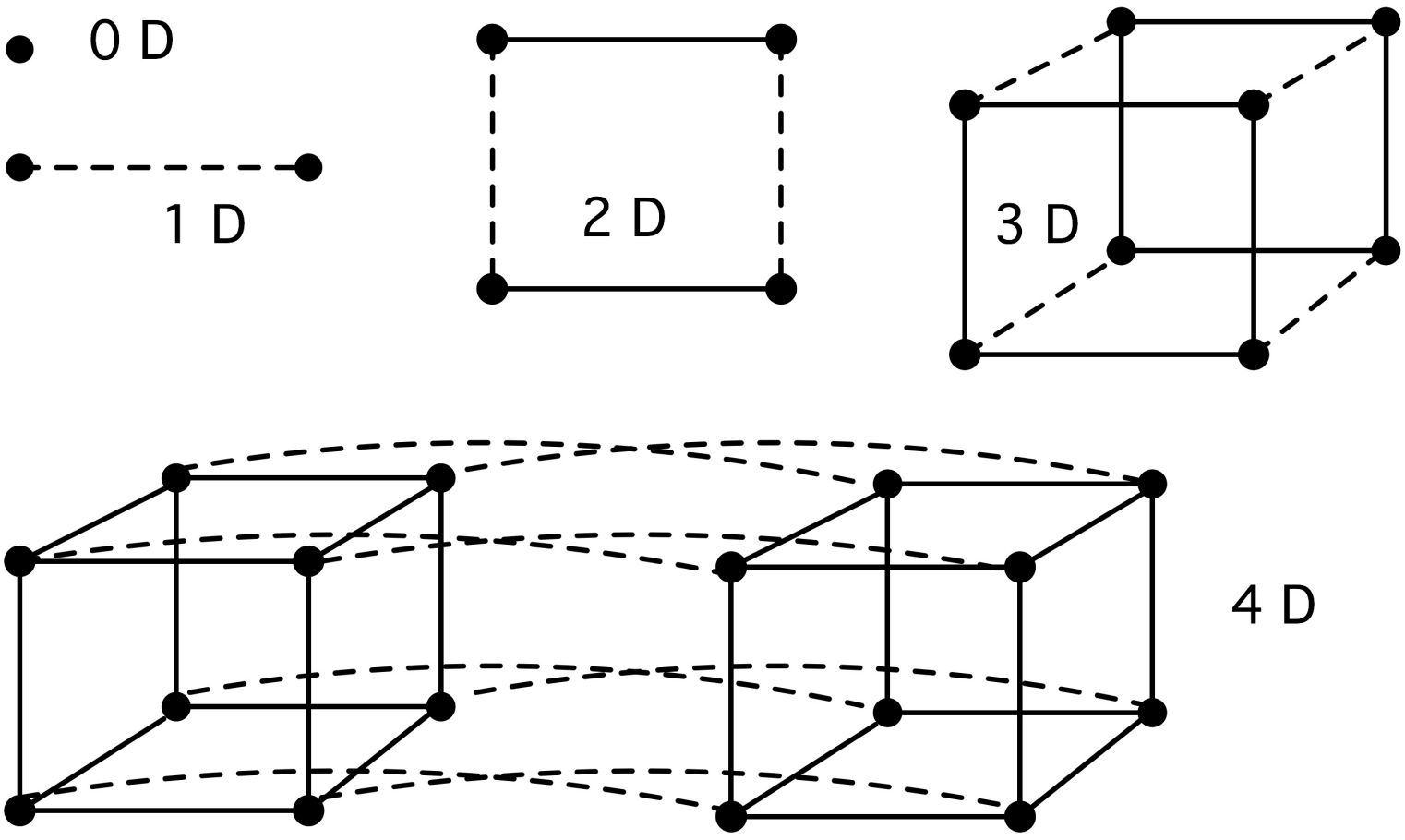
Exercise 3: Broadcast algorithm

Flip last bit, flip one before, . . .

44 Design 3: 2/3-D arrays

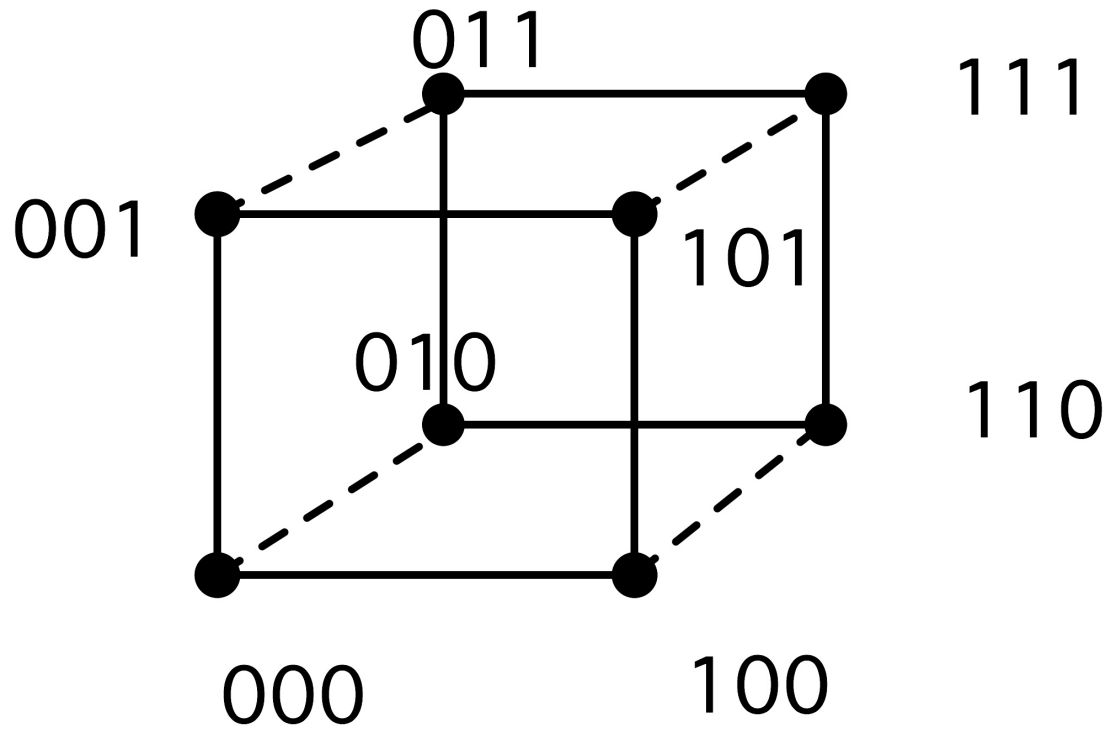
- Degree $2d$, diameter $P^{1/d}$
- Natural design: nature is three-dimensional
- More dimensions: less contention.
  K-machine is 6-dimensional

45 Design 3: Hypercubes



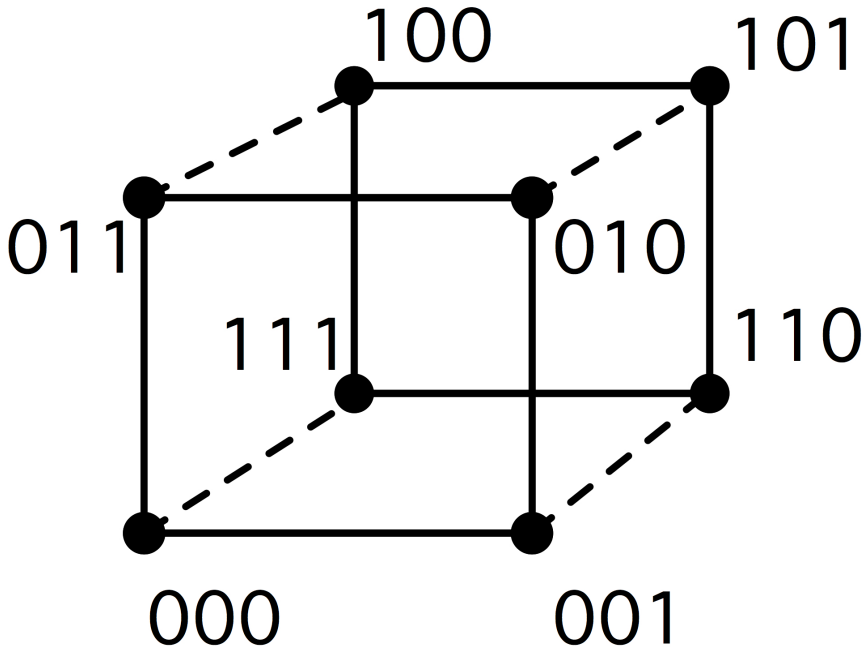
46 Hypercube numbering

Naive numbering:

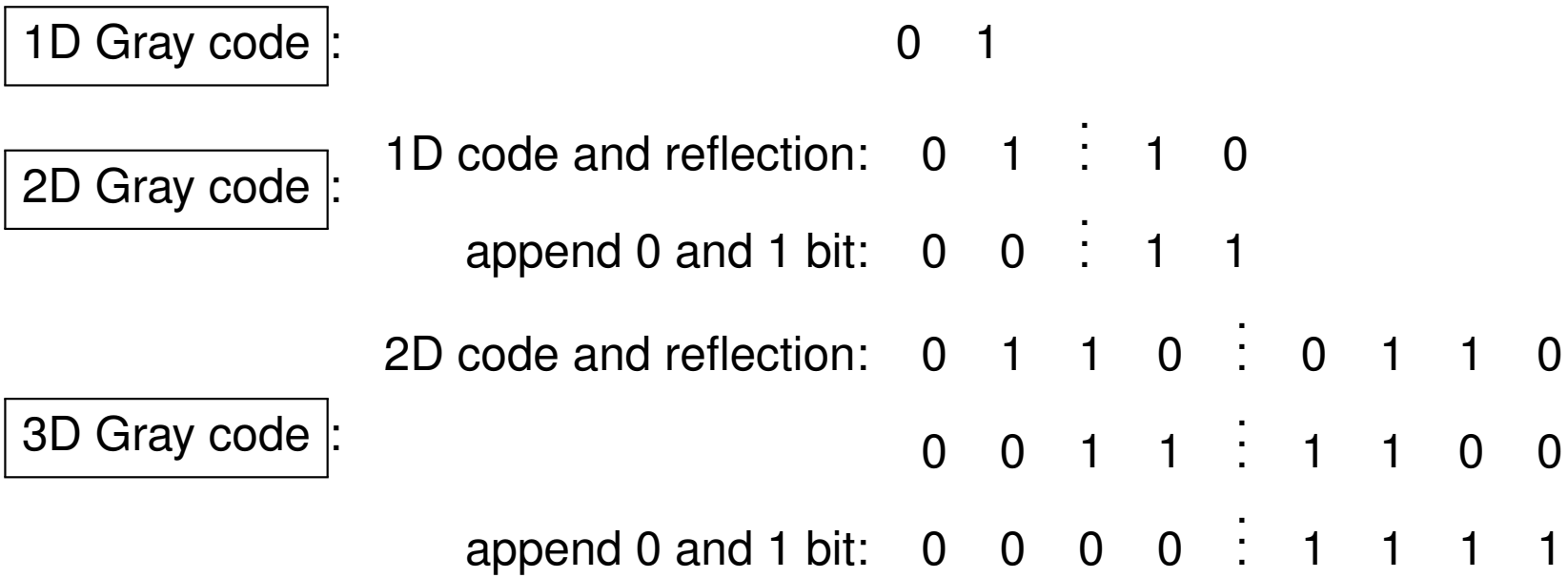


47 Gray codes

Embedding linear numbering in hypercube:



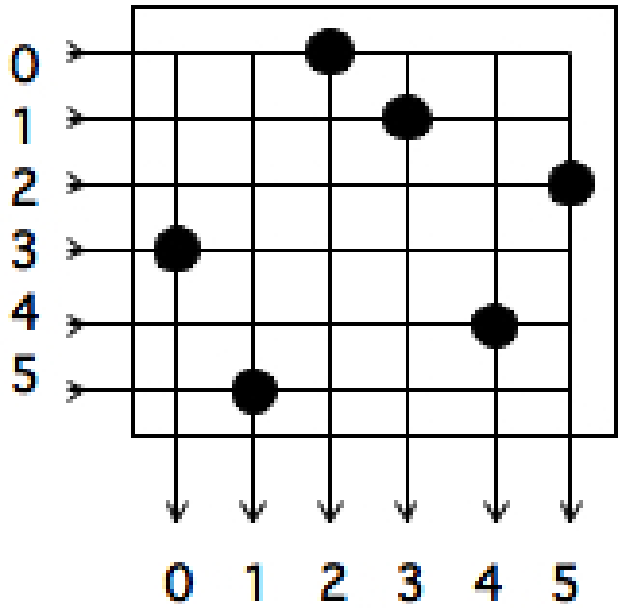
48 Binary reflected Gray code



49 Switching networks

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

50 Cross bar

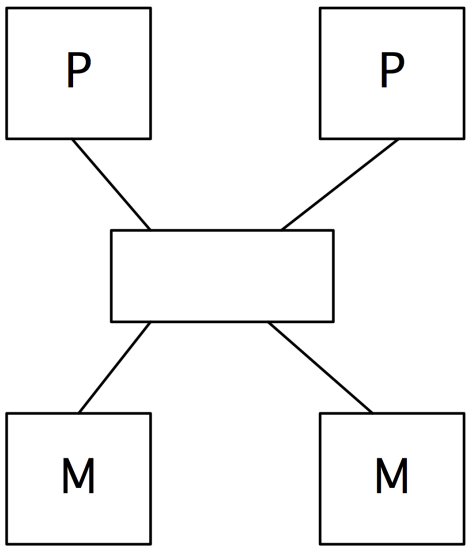


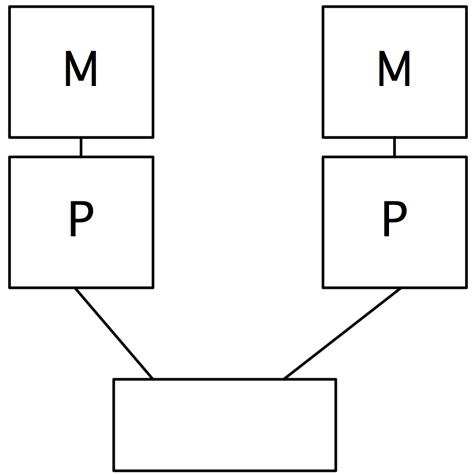
Advantage: non-blocking

Disadvantage: cost

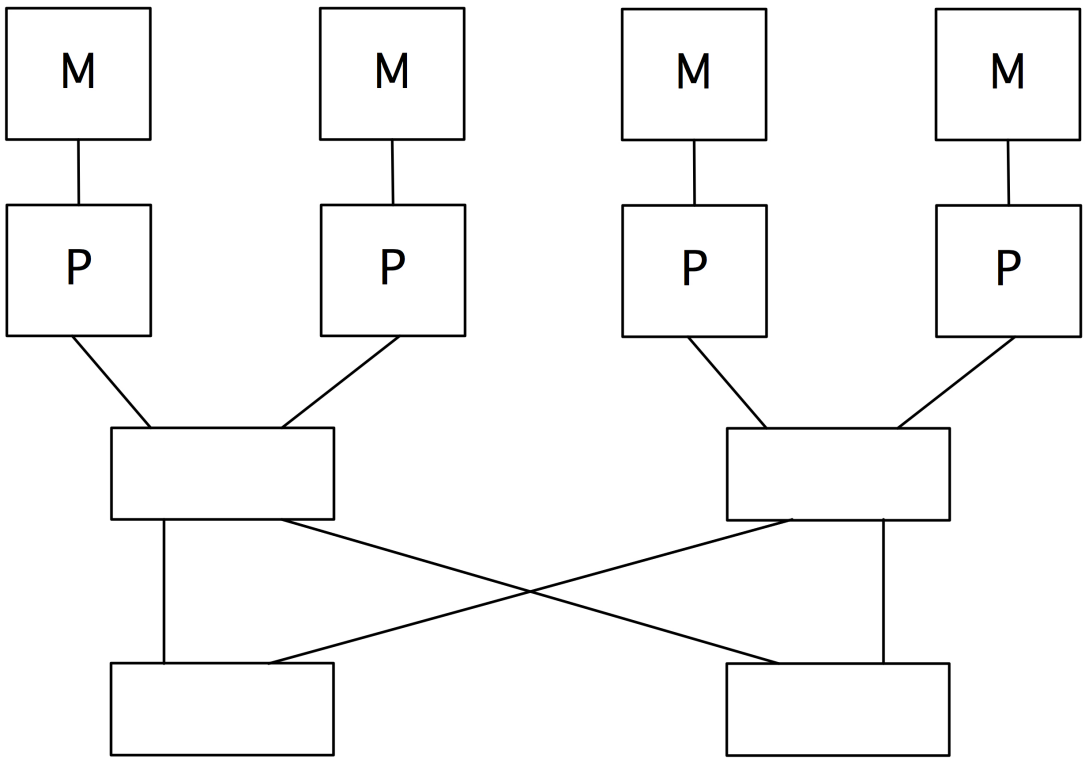
51 Butterfly exchange

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

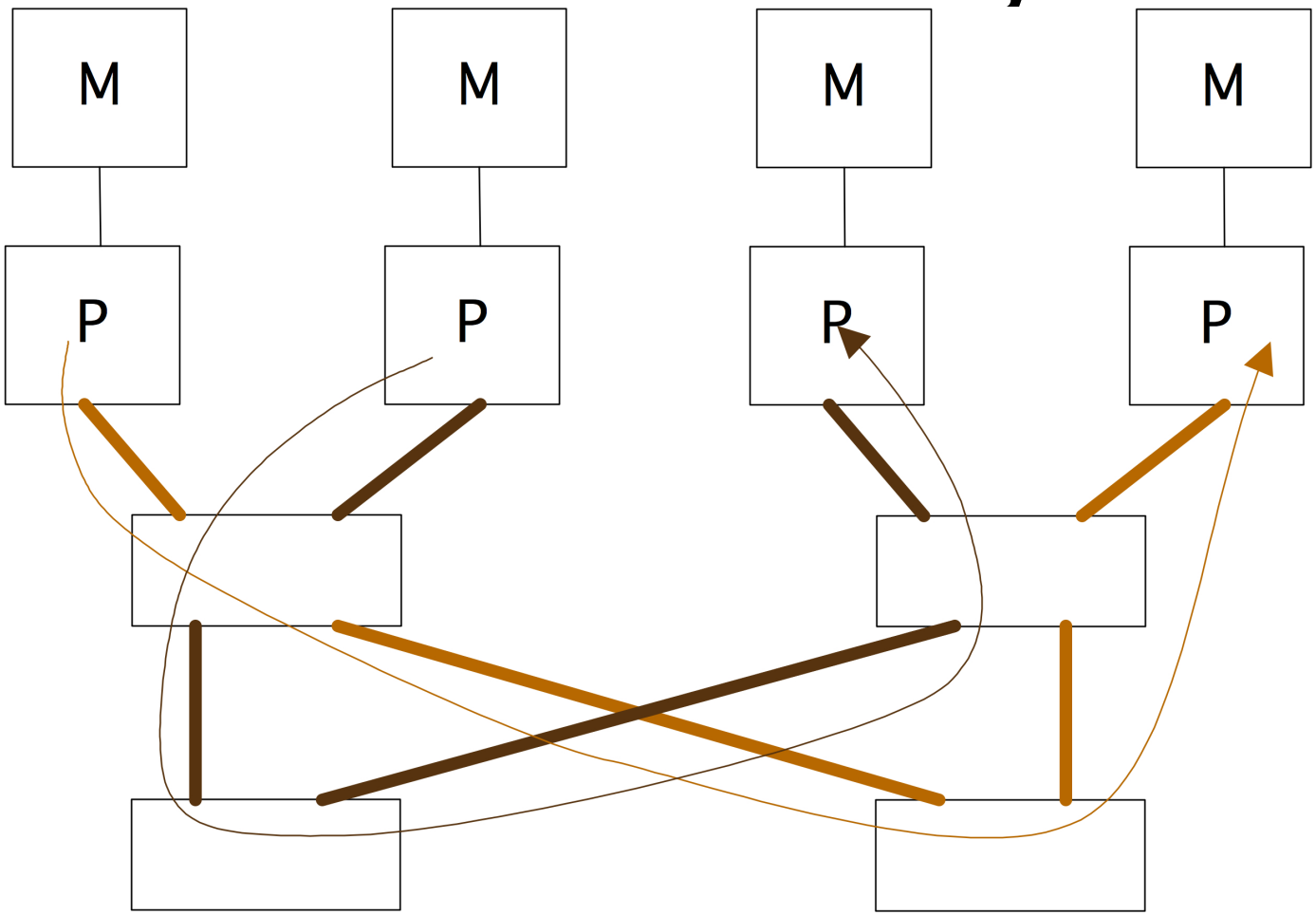




52 Building up butterflies

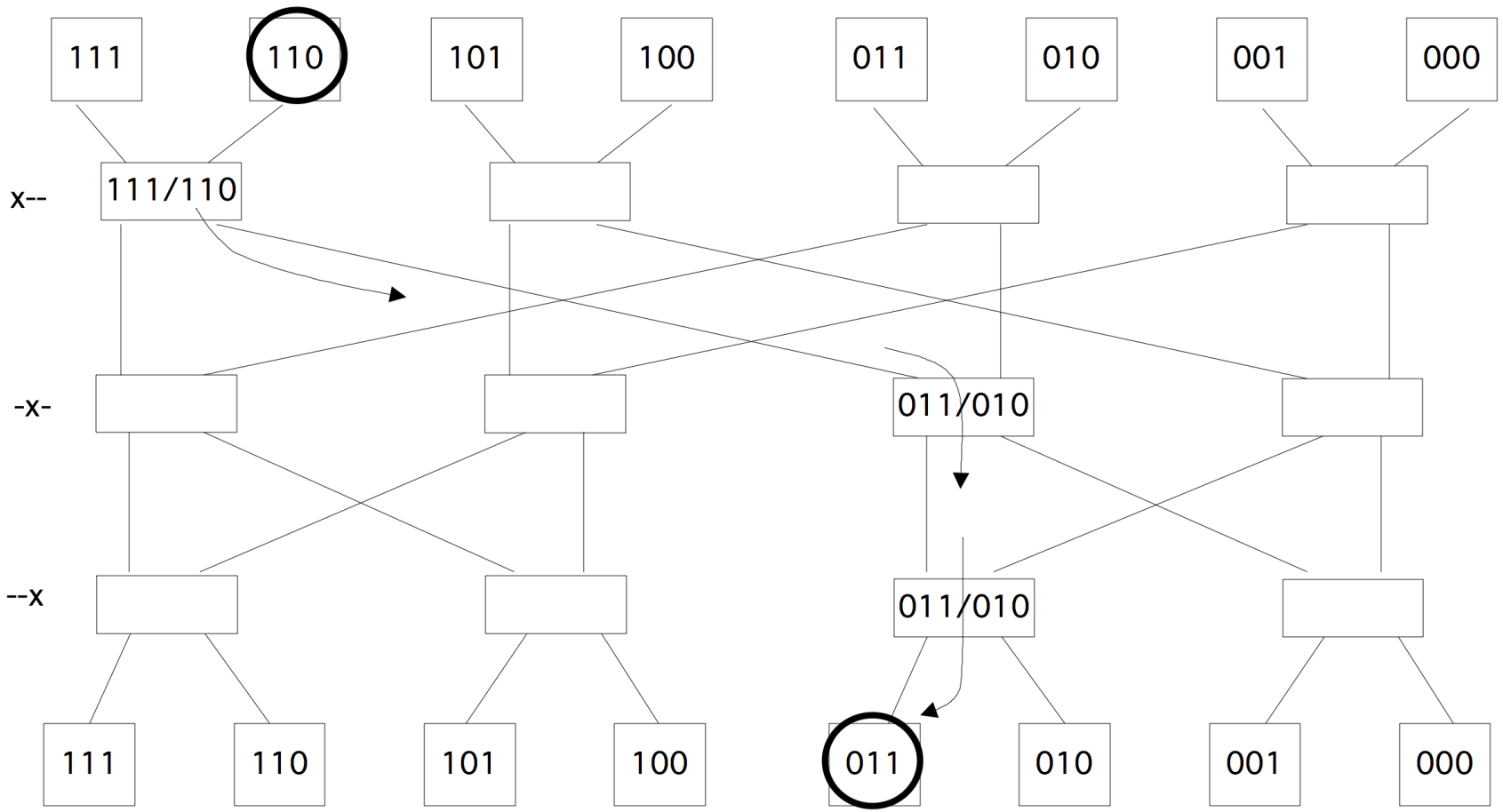


53 Uniform memory access

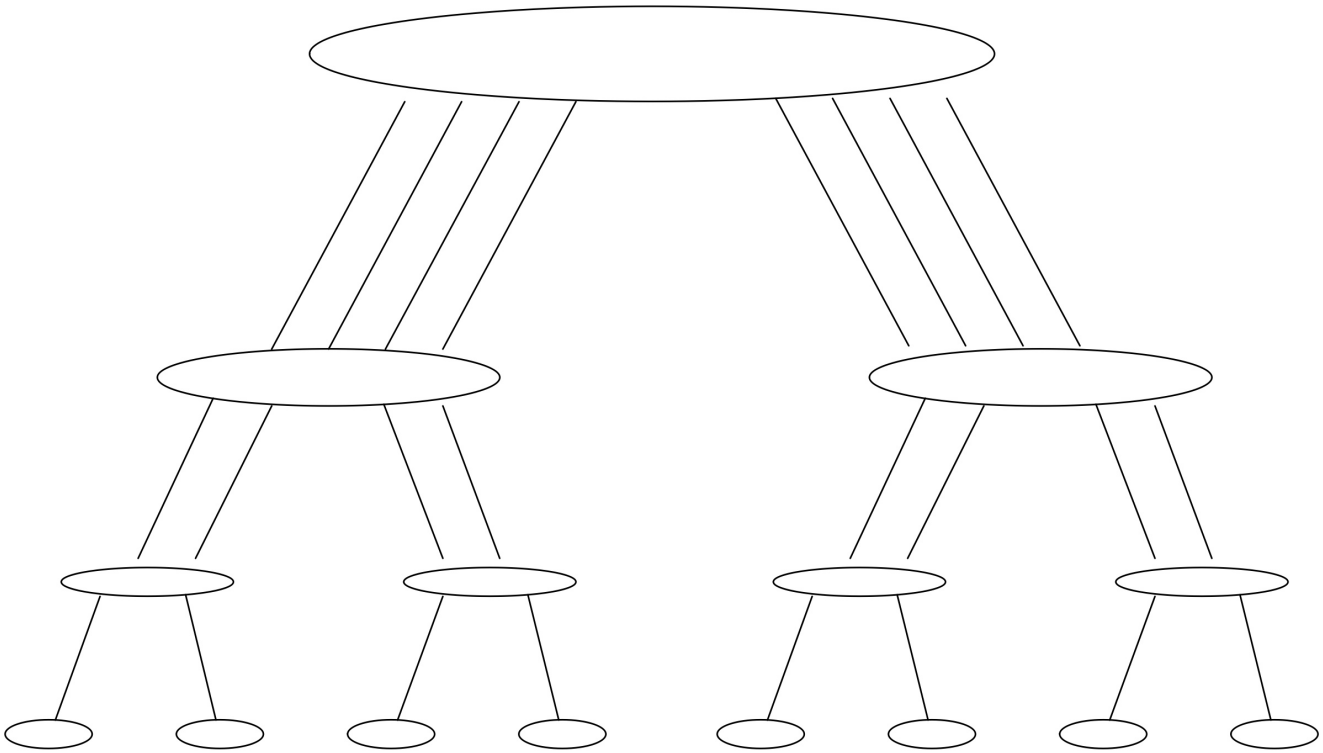


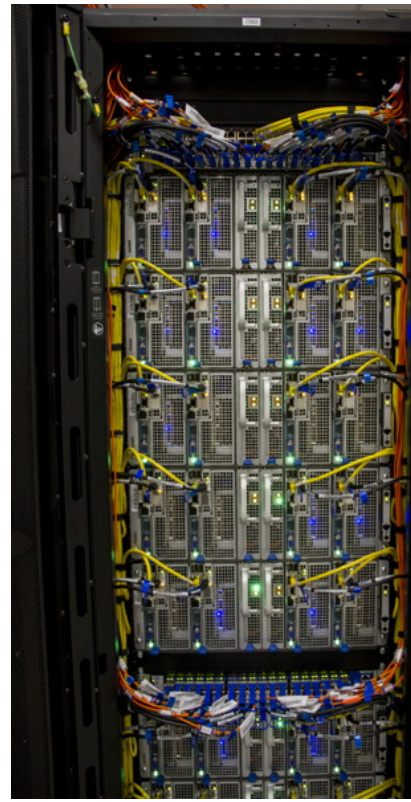
Contention possible

54 Route calculation

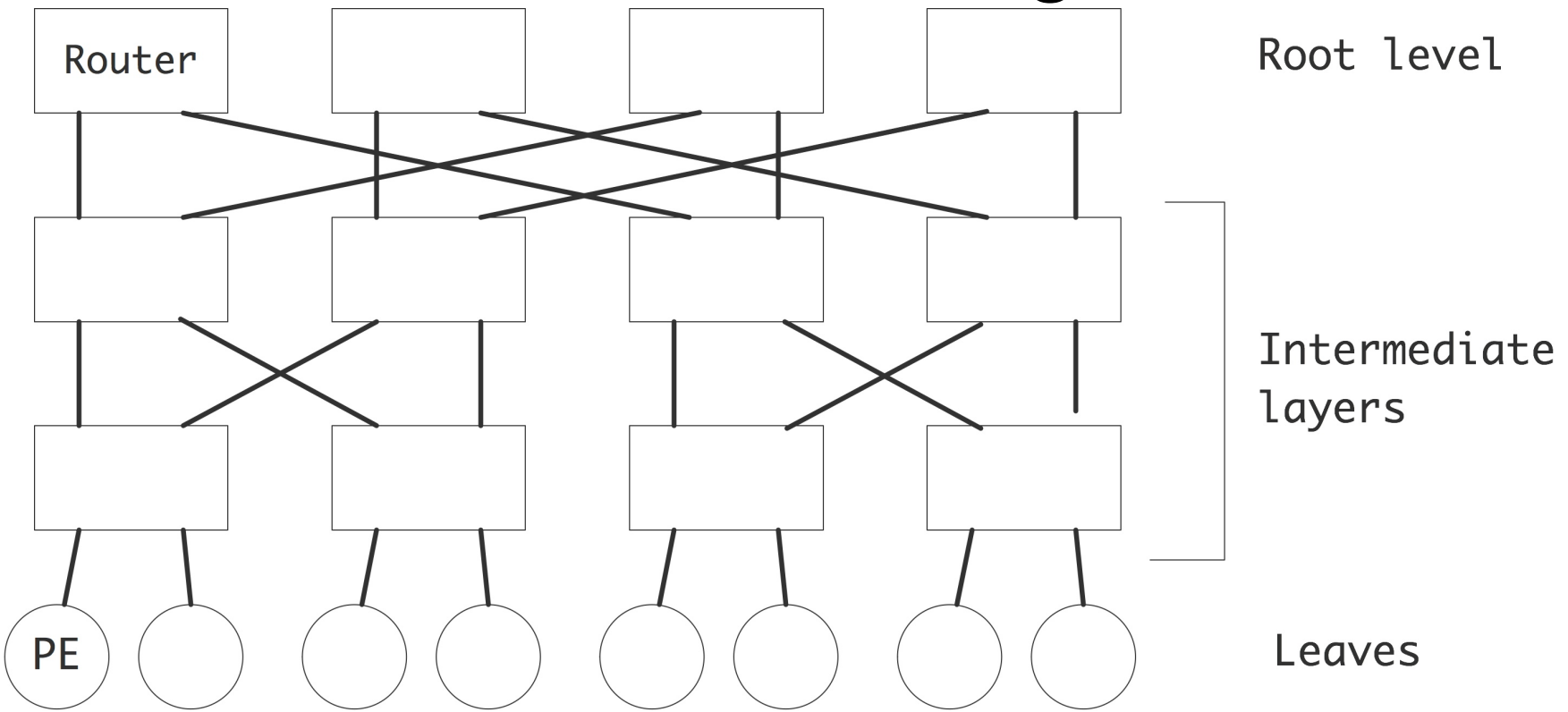


55 Fat Tree





56 Fat trees from switching elements



(Clos network)

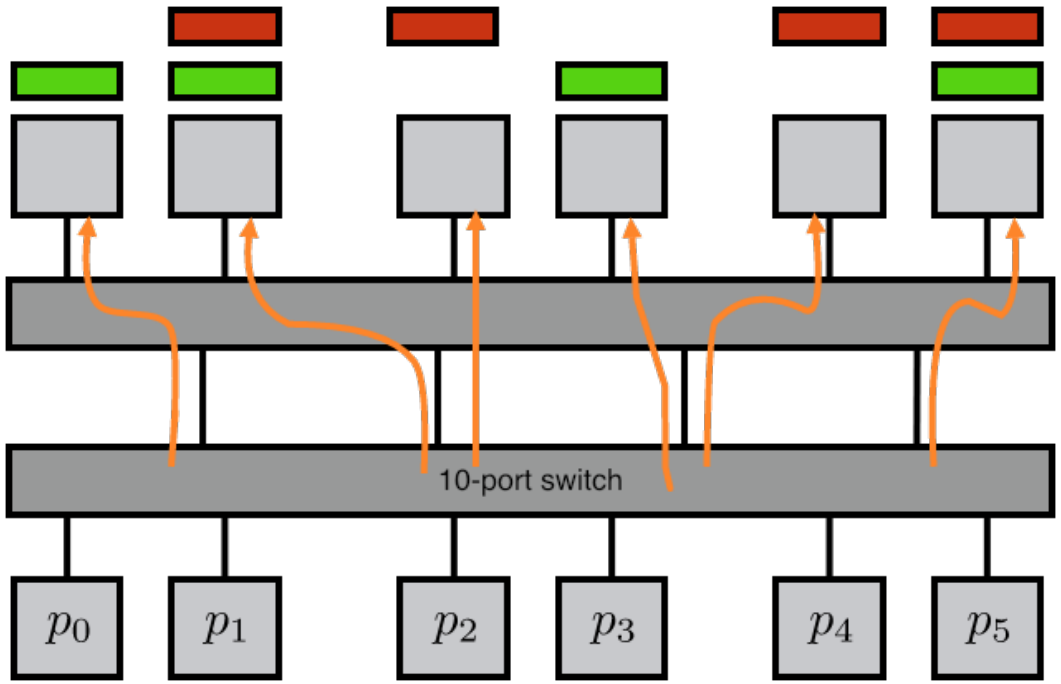
57 Fat tree clusters

Exercise 4: Switch contention

Suppose the number of processor p is larger than the number of wires w.

Write a simulation that investigates the probability of contention if you send $m \leq w$ message to distinct processors.

Can you do a statistical analysis, starting with a simple case?



58 Mesh clusters

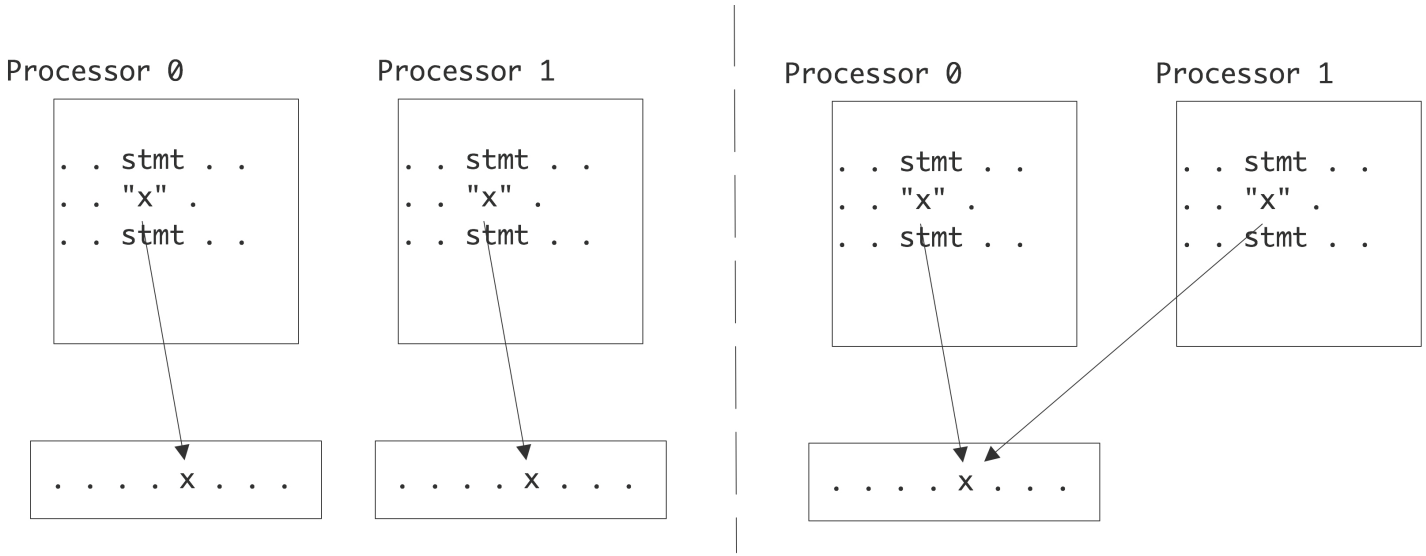
59 Levels of locality

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

Programming models

60 Shared vs distributed memory programming

Different memory models:



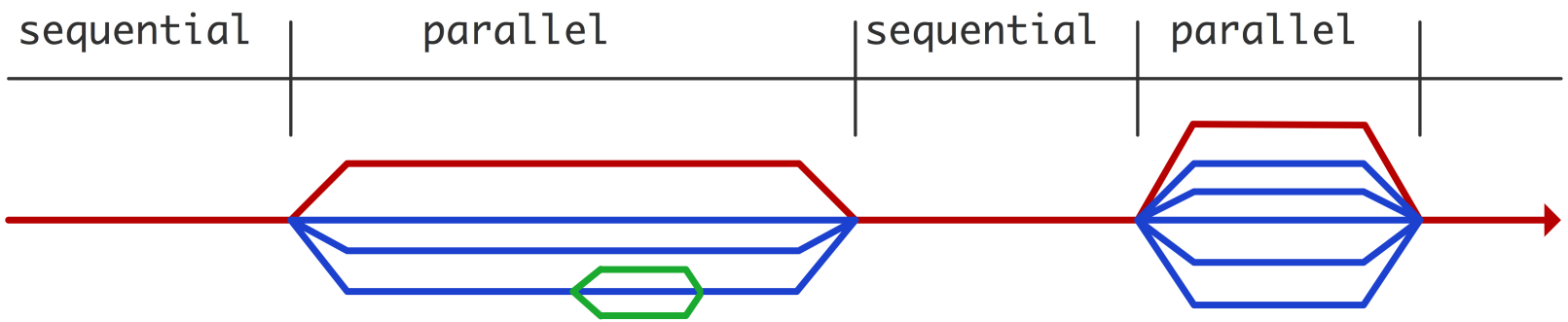
Different questions:

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

Thread parallelism

61 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!

62 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

63 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;
```

64 Race conditions

Init: `I=0`

process 1: `I=I+2`

process 2: `I=I+3`

scenario 1.		scenario 2.		scenari
		$I = 0$		
read $I = 0$	read $I = 0$	read $I = 0$	read $I = 0$	read $I = 0$
compute $I = 2$	compute $I = 3$	compute $I = 2$	compute $I = 3$	compute $I = 2$
write $I = 2$			write $I = 3$	write $I = 2$
	write $I = 3$	write $I = 2$		
$I = 3$		$I = 2$		$I = 5$

65 Dealing with atomic operations

Semaphores, locks, mutexes, critical sections, transactional memory

Software / hardware

66 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

67 OpenMP

- Directive based
- Parallel sections, parallel loops, tasks

Distributed memory parallelism

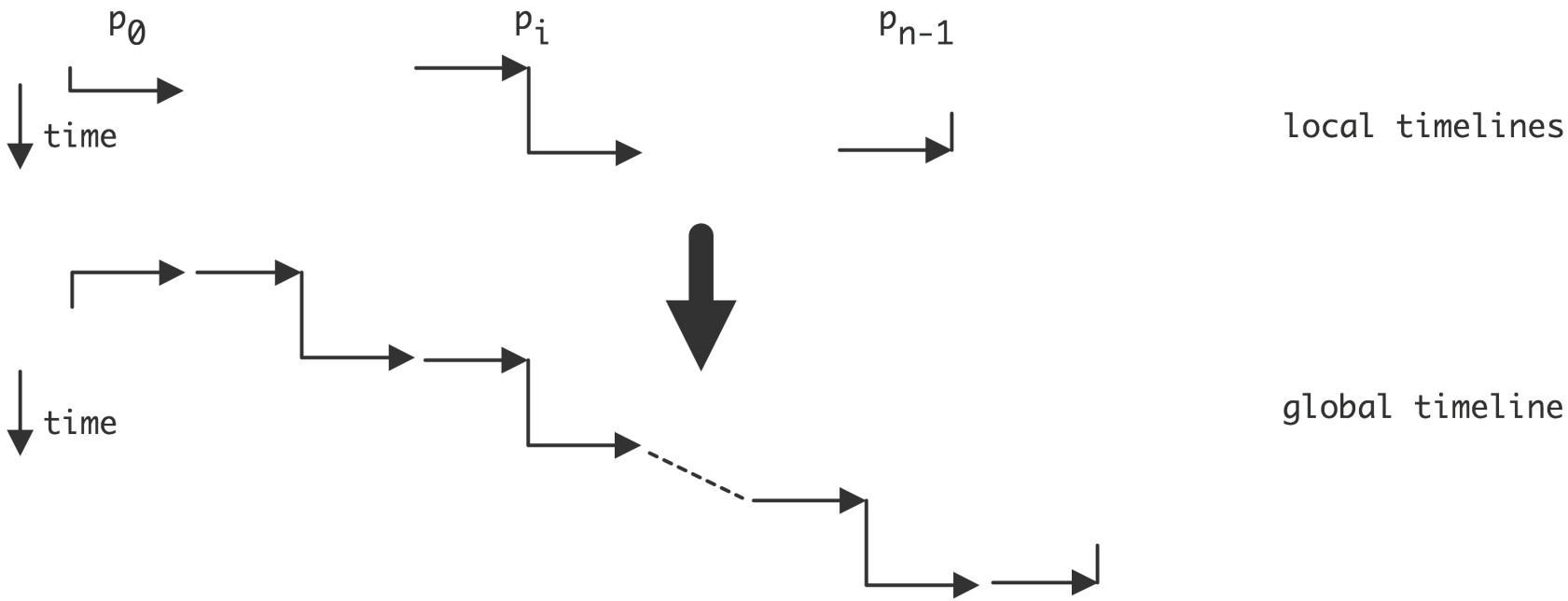
68 Global vs local view

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

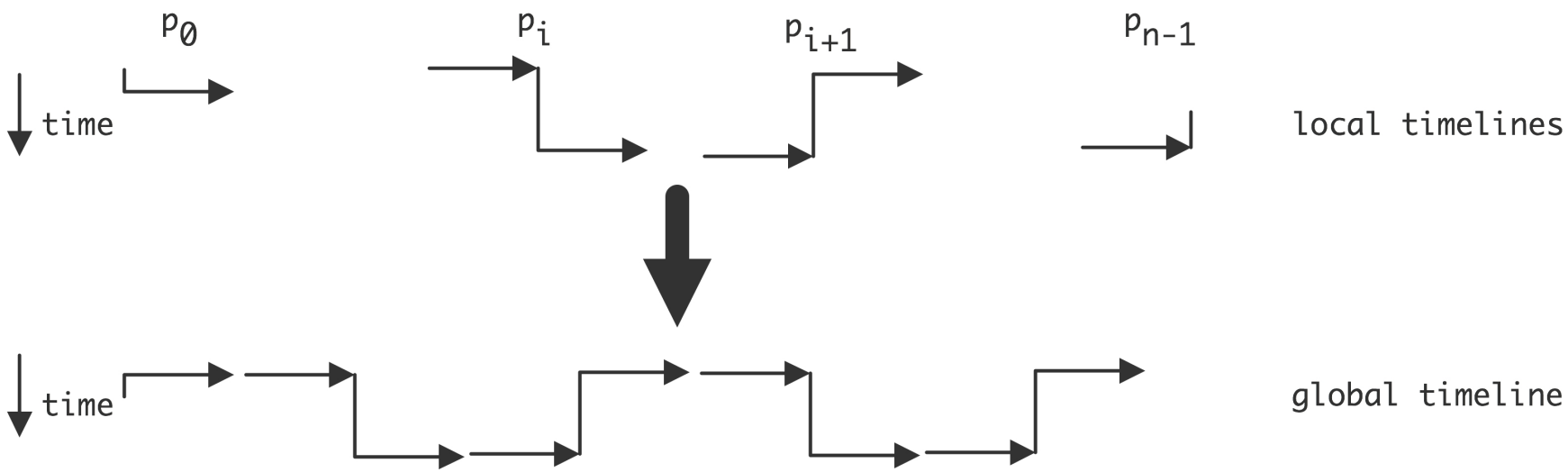
- If I am processor 0 do nothing, otherwise receive a y element from the left, add it to my x element.
- If I am the last processor do nothing, otherwise send my y element to the right.

(Let's think this through. . .)

69 Global picture



70 Careful coding



71 Better approaches

- Non-blocking send/receive
- One-sided

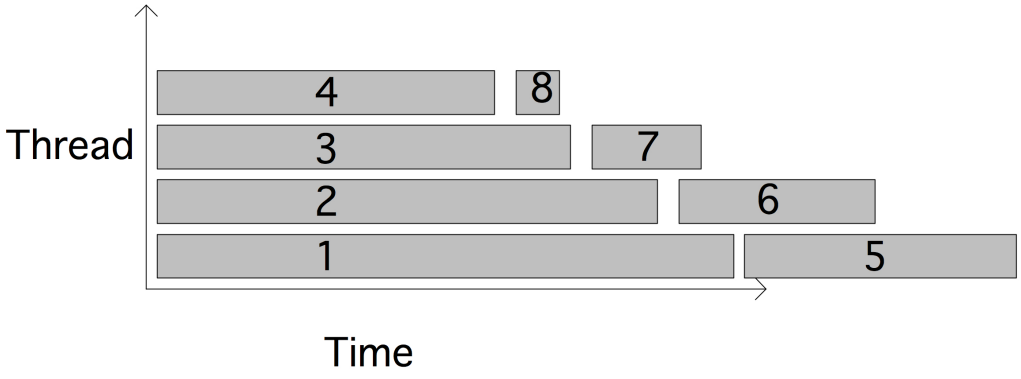
Hybrid/heterogeneous parallelism

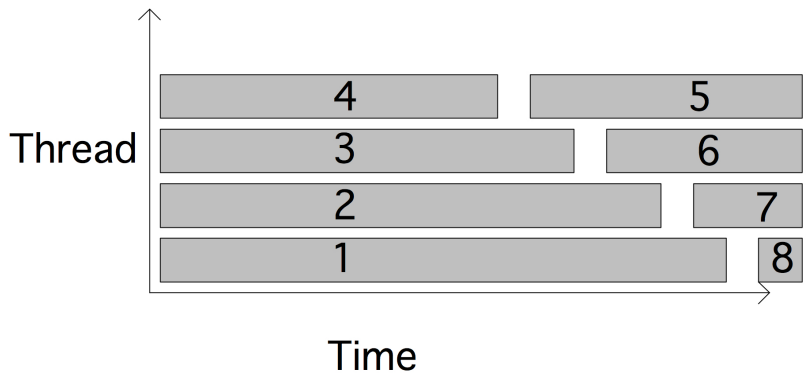
72 Hybrid computing

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

73 Using threads for load balancing

Dynamic scheduling gives load balancing





Hybrid is possible improvement over strict-MPI

74 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:

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

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

Design patterns

75 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)
     );
```

76 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);
}
```

77 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;
}
```

78 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

79 Latency hiding

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

80 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

81 Parallel languages

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

82 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];
}
```

83 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

84 Grab bag of other approaches

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

Load balancing, locality, space-filling curves

85 The load balancing problem

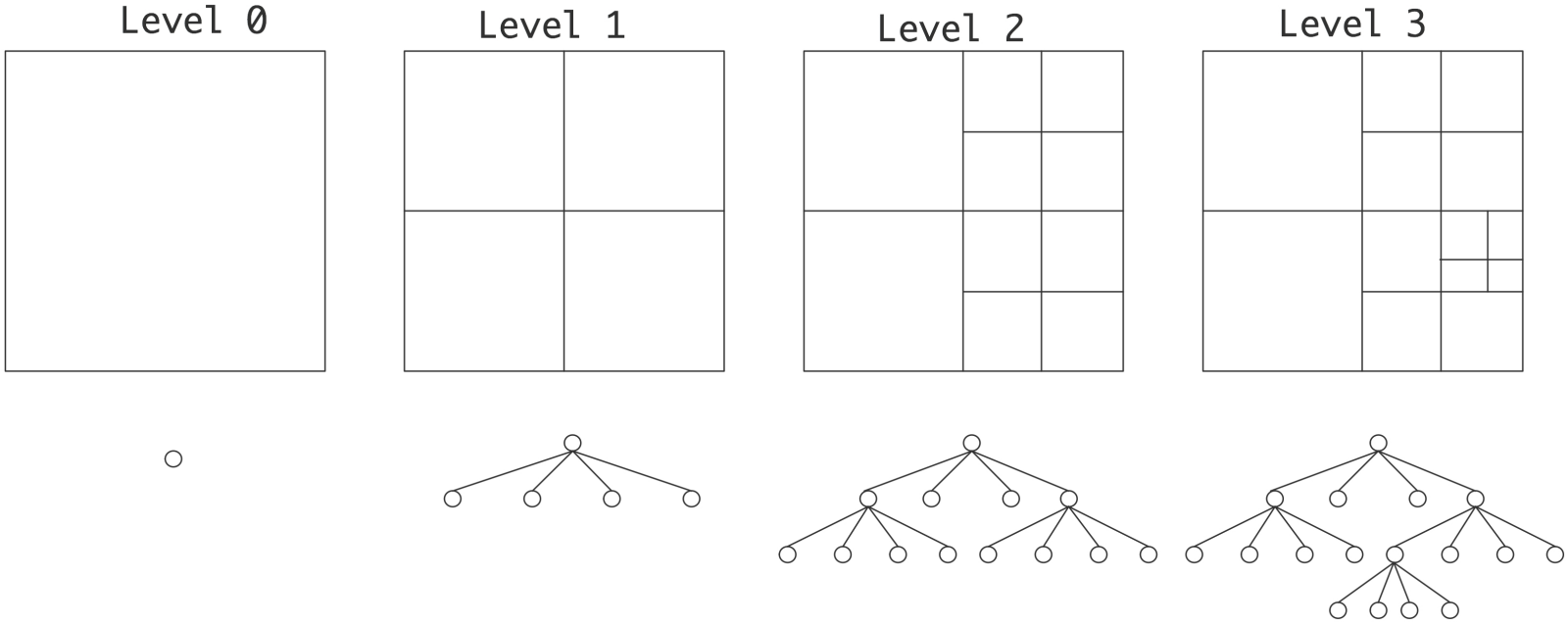
- Application load can change dynamically
  e.g., mesh refinement, time-dependent problems
- Splitting off and merging loads
- No real software support: write application anticipating load management
- Initial balancing: graph partitioners

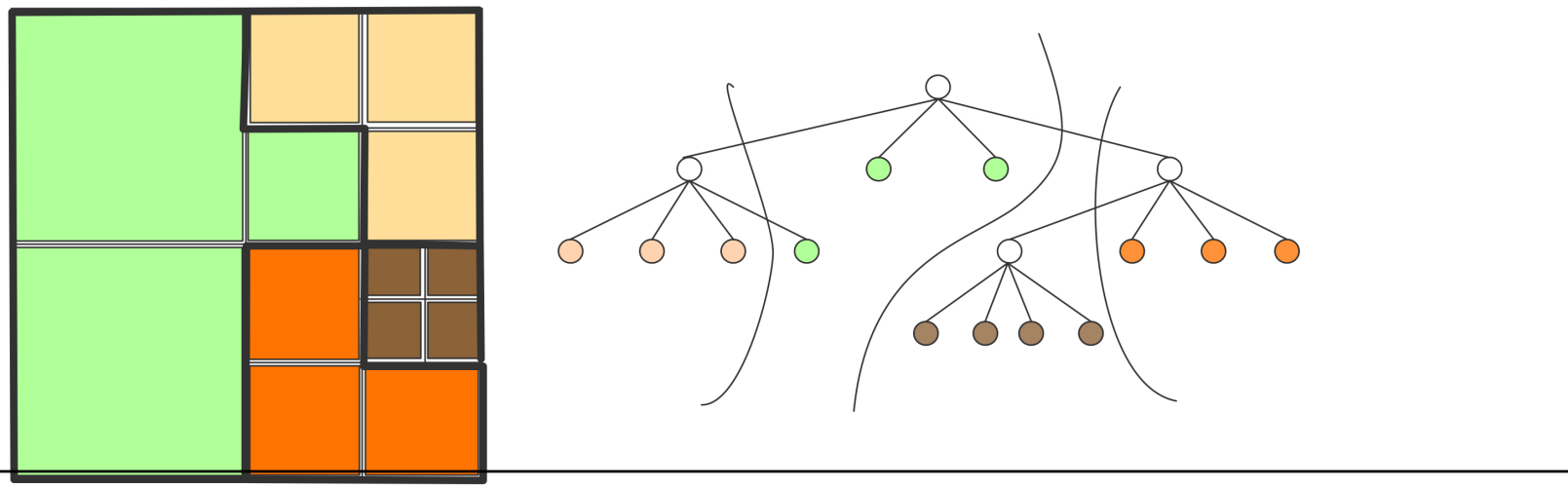
86 Load balancing and performance

- 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?

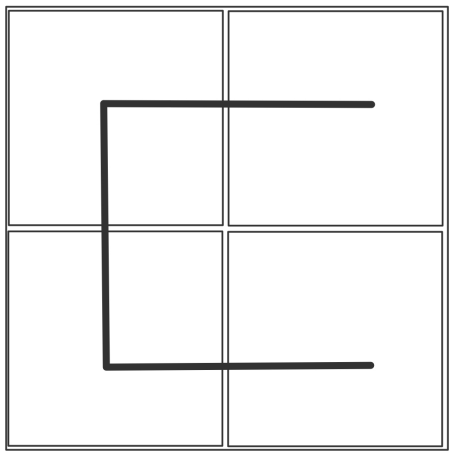
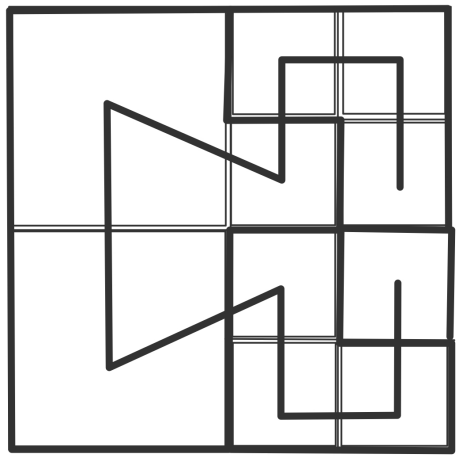
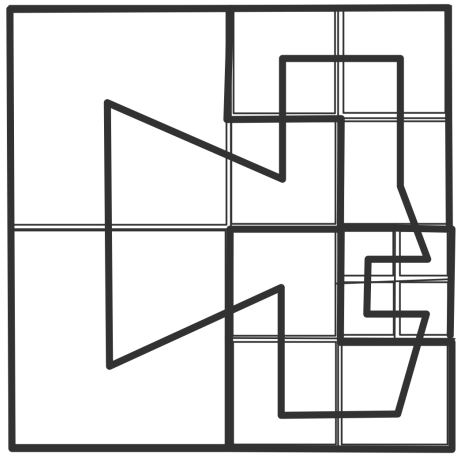
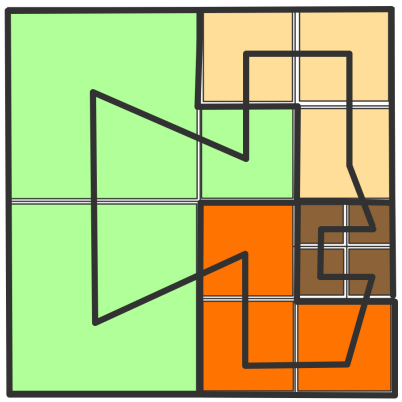
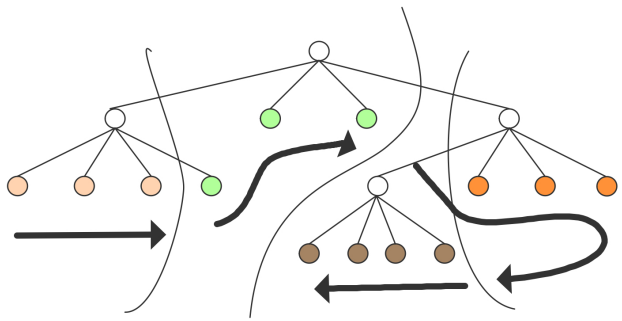
Space-filling curves

87 Adaptive refinement and load assignment





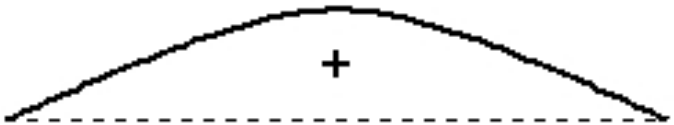
88 Assignment through Space-Filling Curve

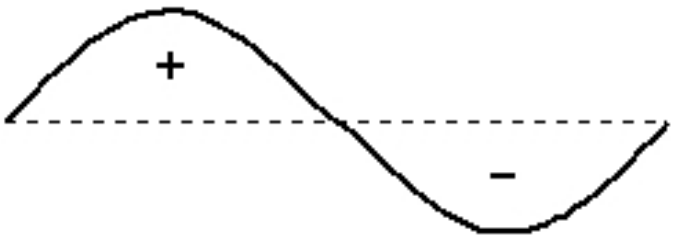
Domain partitioning by Fiedler vectors

89 Inspiration from physics

Modes of a Vibrating String



Lowest Frequency lambda(1)



Second Frequency lambda(2)

Third Frequency lambda(3)

90 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

91 Fiedler in a picture