What is parallelism?

Victor Eijkhout

Fall 2025

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.

Parallelism

Basic concepts

The basic idea

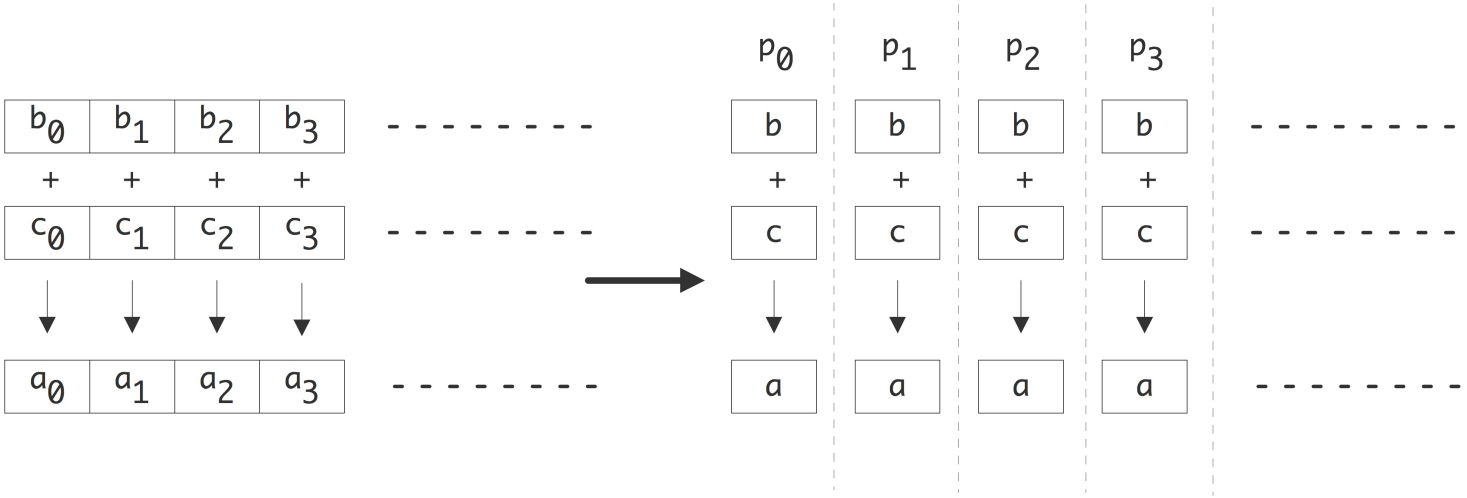
Parallelism is about doing multiple things at once.

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

Simple example

Summing two arrays together:

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



Parallel: every processing element does

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

Time goes down linearly with processors

Differences between operations

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

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

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

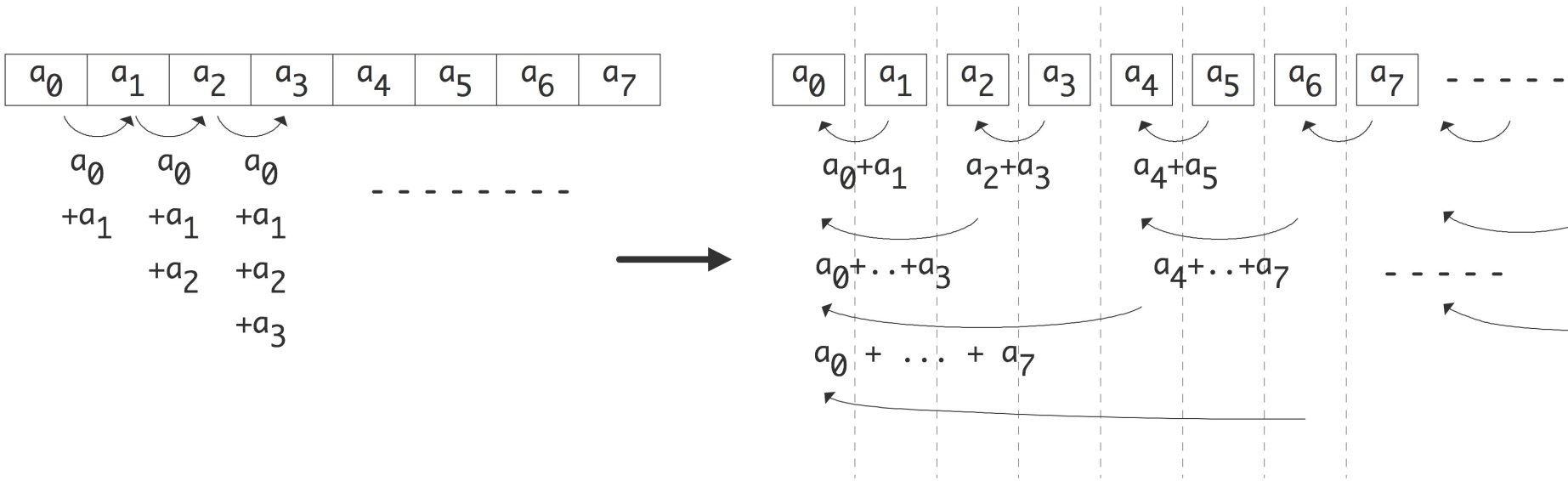
Summing

Naive algorithm

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

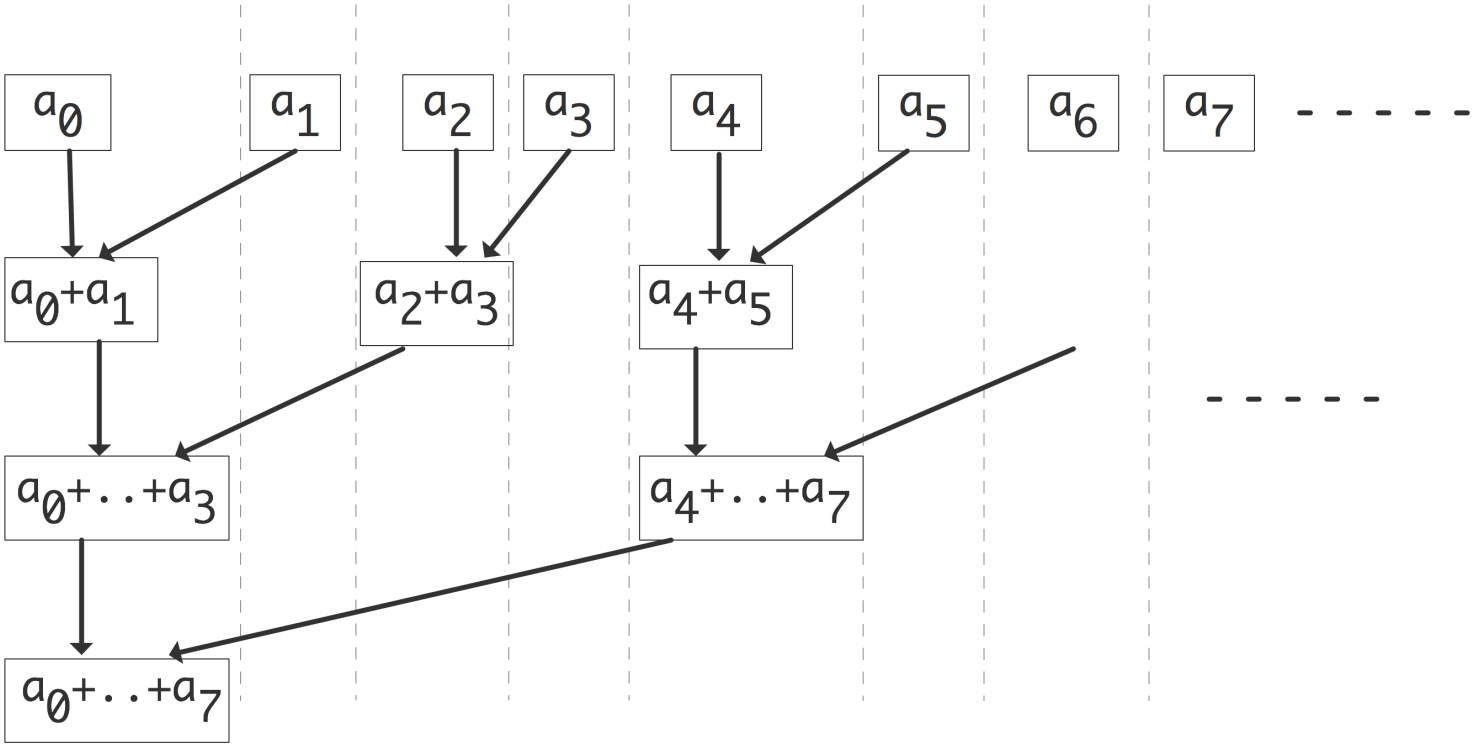
Recoding

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



And then there is hardware

Topology of the processors:



increasing distance: limit on parallel speedup

Theoretical concepts

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

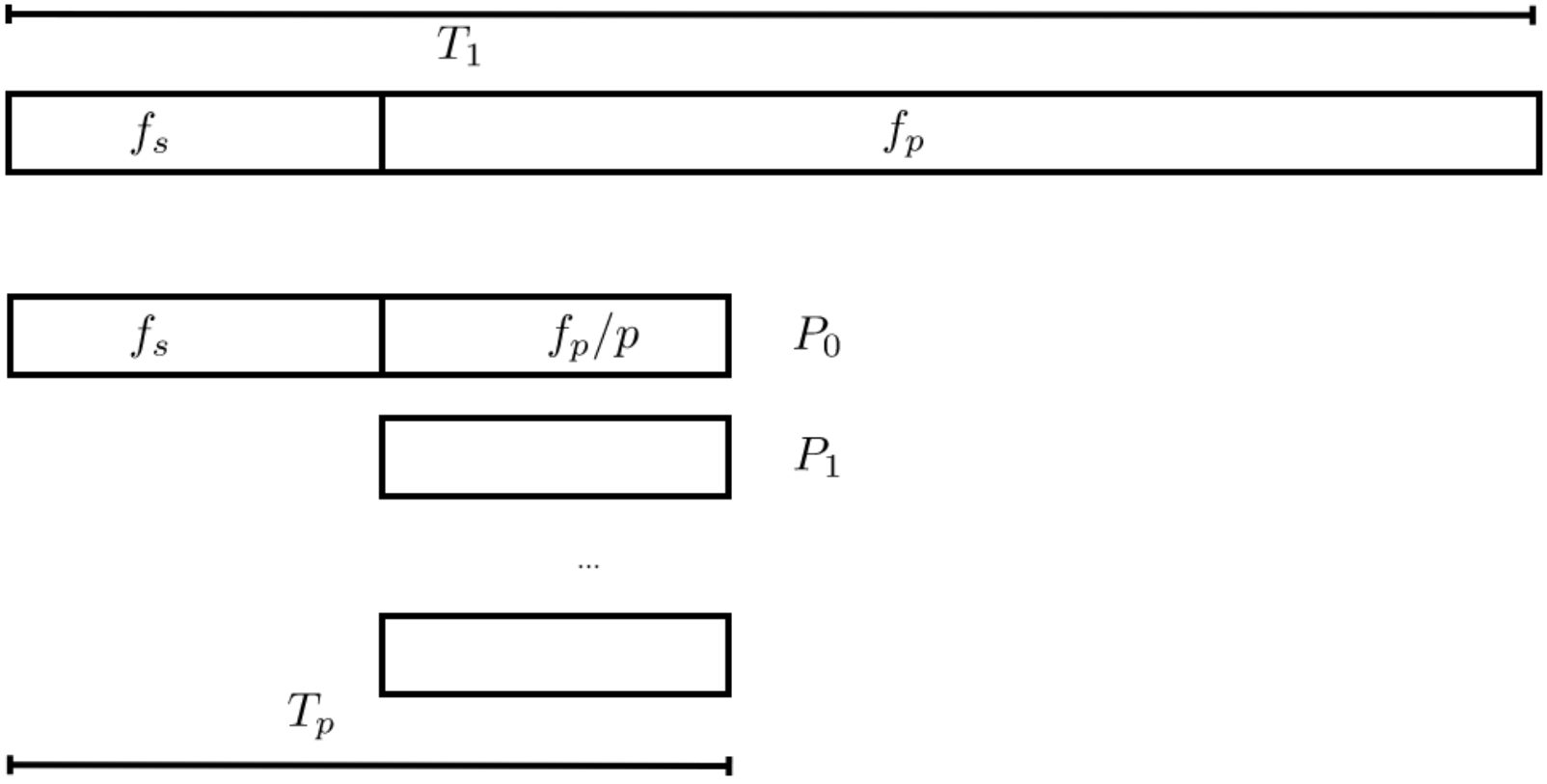
But:

- 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?

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
- $F_s + F_p = 1$



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 $E \sim 1/P$.

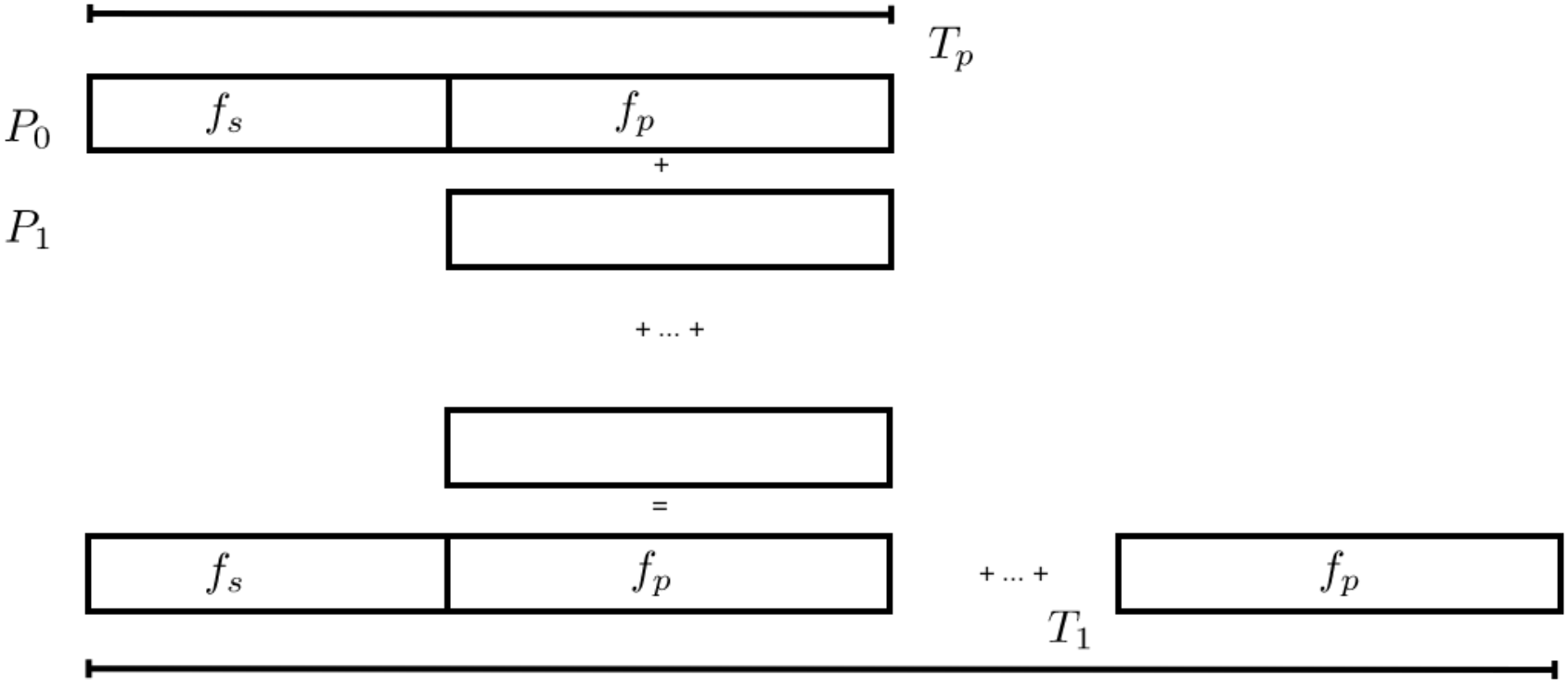
Do you see problems with this?

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

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



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

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 $E_p = E(P, M)$.)

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 $E_p = E(P, M)$.)

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:

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

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.

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

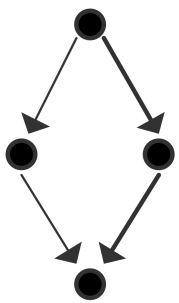
- Substituting $M = Pm$, we find ultimately

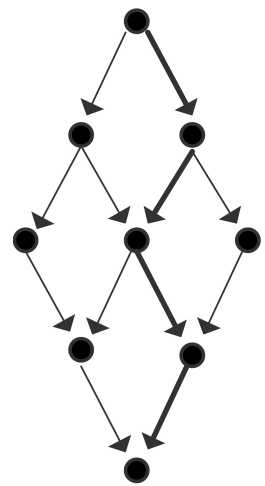
$$m = C \begin{cases} 1 \,/\, P^{1/(d+1)} & \text{hyperbolic} \\ 1 \,/\, P^{2/(d+2)} & \text{parabolic} \end{cases}$$

Critical path analysis

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.





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.

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

```
1 for (i=0; i<1000000; i++)
2   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

Unsynchronized tasks: fork-join

general scheduler

Conveniently parallel

Example: Mandelbrot set
Parameter sweep,
often best handled by external tools

Medium-grain parallelism

Mix of data parallel and task parallel

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

LU factorization analysis

Algorithm

for $k = 1, n-1$:
    for $i = k+1$ to n:
        $a_{ik} \leftarrow a_{ik} / a_{kk}$
    for $i = k+1$ to n:
        for $j = k+1$ to n:
            $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

Can the k loop be done in parallel? The i, j loops?

Dependent operations

$$a_{22} \leftarrow a_{22} - a_{21} * a_{11}^{-1} a_{12}$$
$$\ldots$$
$$a_{33} \leftarrow a_{33} - a_{32} * a_{22}^{-1} a_{23}$$

Exercise 1

Critical path Follow this argument through. Argue that there is a non-trivial critical path in the sense of section **??**. What is its length? In the analysis of the critical path section, what does this critical path imply for the minimum parallel execution time and bounds on speedup?

Subblock update

for $i = k + 1$ to n:
    for $j = k + 1$ to n:
        $a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

How many processors can you use maximally in step k?

Exercise 2

Parallel execution Continue this reasoning. With $p = n^2$ processing elements each of the (i, j) updates in the subblock can be done simultaneously. To be precise, how long does an arbitrary k iteration take? Summing over all k, what is the resulting T_p, S_p, E_p? How does this relate to the bounds you derived above?
Also, with $p = n$ processing elements you could let each row or column of the subblock update be done in parallel. What is now the time for the kth outer iteration? What is the resulting T_p, S_p, E_p?,

Application scaling

Single processor.

Relating time and memory to problem size

$$T = \frac{1}{3} N^3 / f, \qquad M = N^2.$$

where f is processor frequency.

Exercise 3

Memory scaling, case 1: Faster processor Suppose 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?

More processors

Keep frequency constant, but vary number of processors p:

$$T = \frac{1}{3} N^3 / p, \qquad M = N^2.$$

Each processor now stores $M_p = N^2 / p$ elements.

Exercise 4

Memory scaling, case 2: More processors Suppose you have a cluster with p processors, each with M_p memory, can run a Gaussian elimination of an $N \times N$ matrix in time T:

$$T = \frac{1}{3} N^3 / p, \qquad M_p = N^2 / p.$$

Now you extend the cluster to $2P$ processors, of the same clock speed, and you want to do a benchmark run, again taking time T. How much memory does each node need?
Hint: for the extended cluster:

$$T' = \frac{1}{3} N'^3 / p', \qquad M'_p = N'^2 / p'.$$

The question becomes to compute M'_p under the given conditions.

The SIMD/MIMD/SPMD/SIMT model for parallelism

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

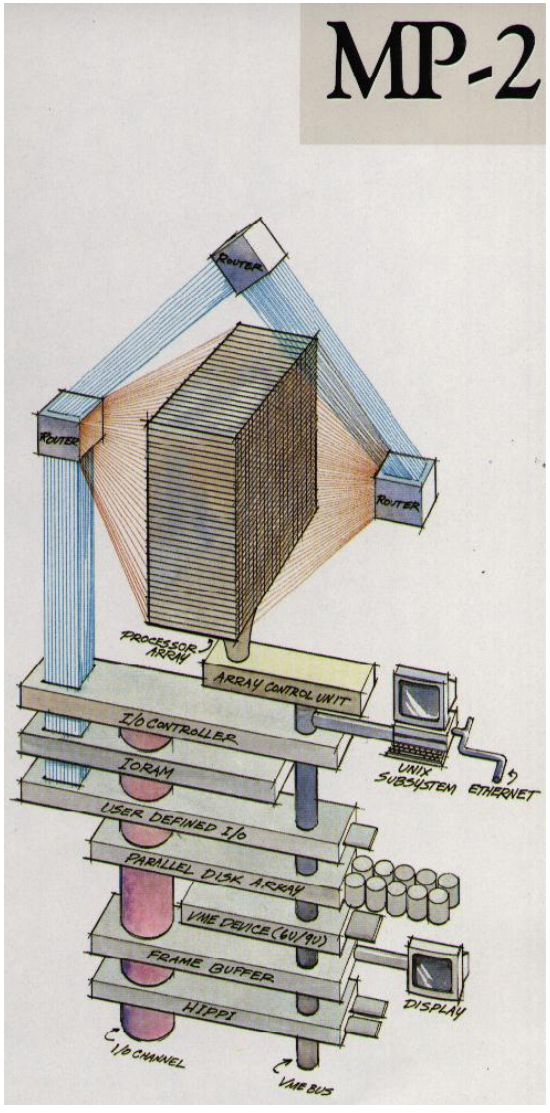
SIMD

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

SIMD: array processors

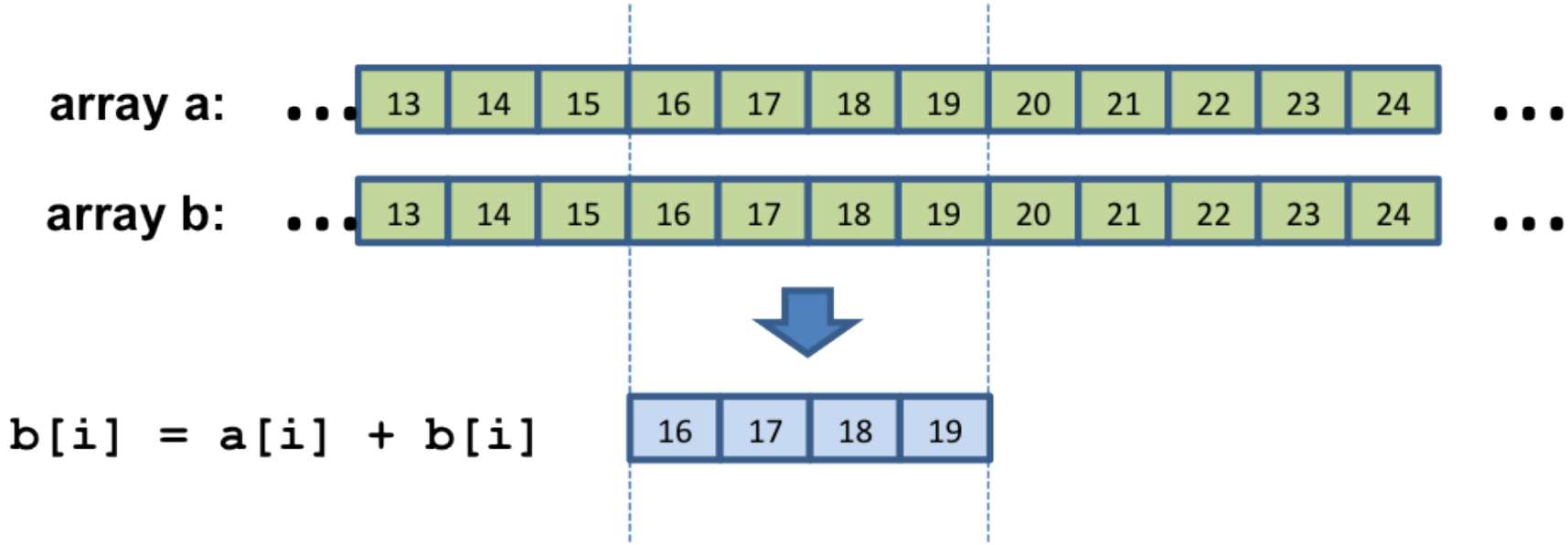
Technology going back to the 1980s: FPS, MasPar, CM, GoodYear

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

```
1 void func(float *restrict c, float *restrict a,
2           float *restrict b, int n)
3 {
4 #pragma vector always
5   for (int i=0; i<n; i++)
6     c[i] = a[i] * b[i];
7 }
```

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

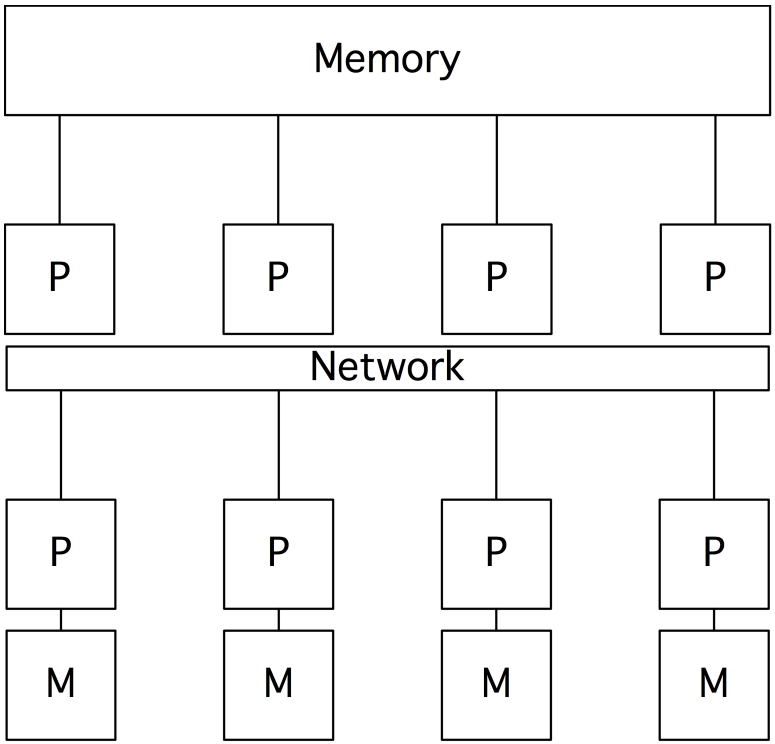
- MIMD: independent processors, independent instruction streams, independent data
- In practice very little true independence: usally the same executable
  Single Program Multiple Data
- Exceptional example: climate codes
- Old-style SPMD: cluster of single-processor nodes
- New-style: cluster of multicore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

GPUs and data paralleism

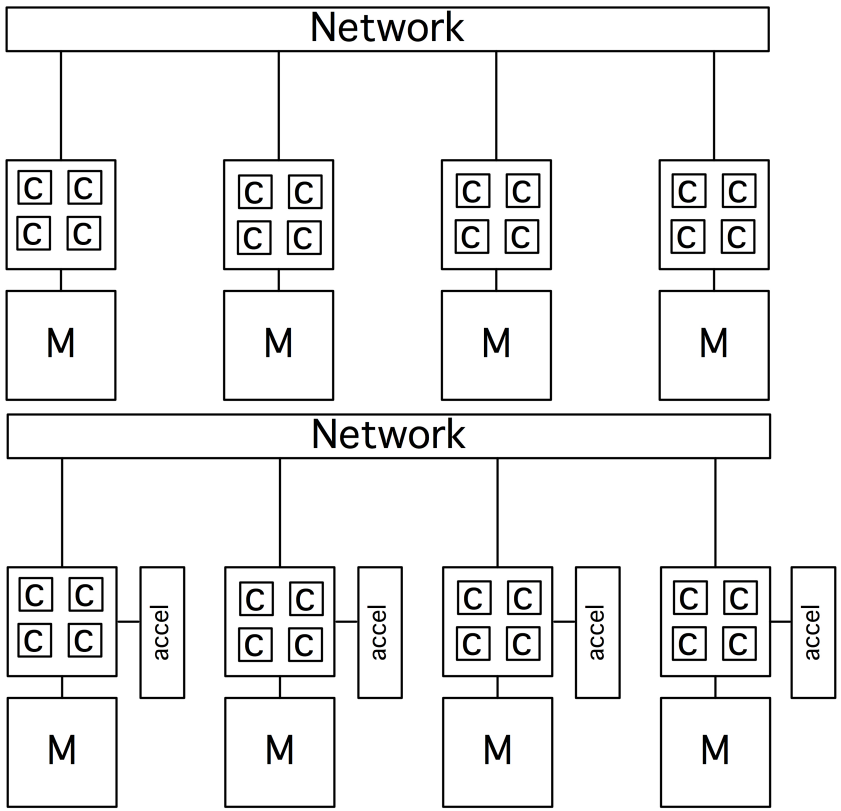
Lockstep in thread block,
single instruction model between streaming processors
(more about GPU threads later)

Characterization of parallelism by memory model

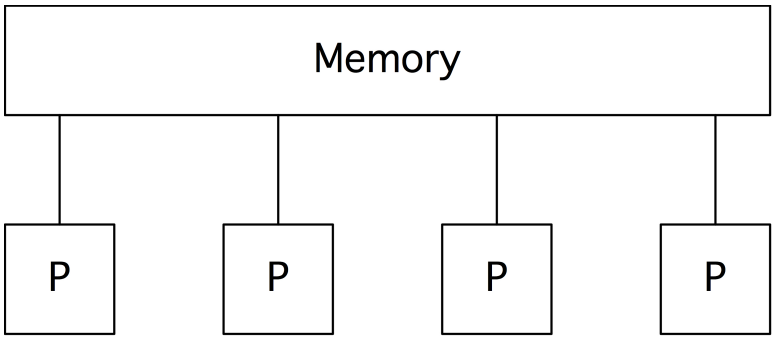
Major types of memory organization, classic



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

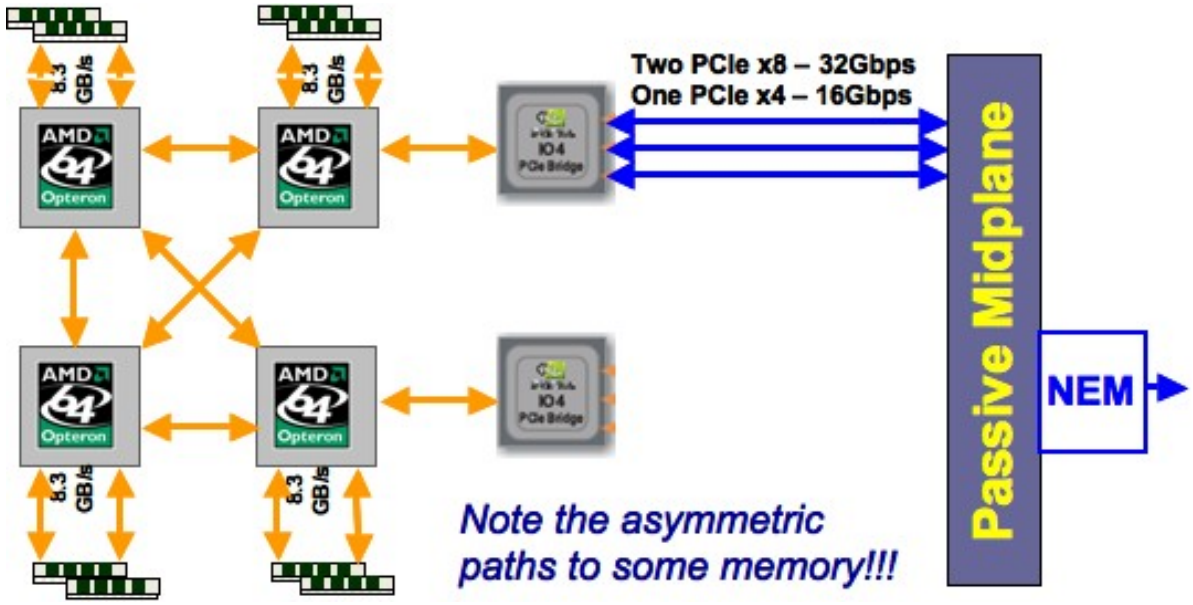
SMP, bus design

- 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

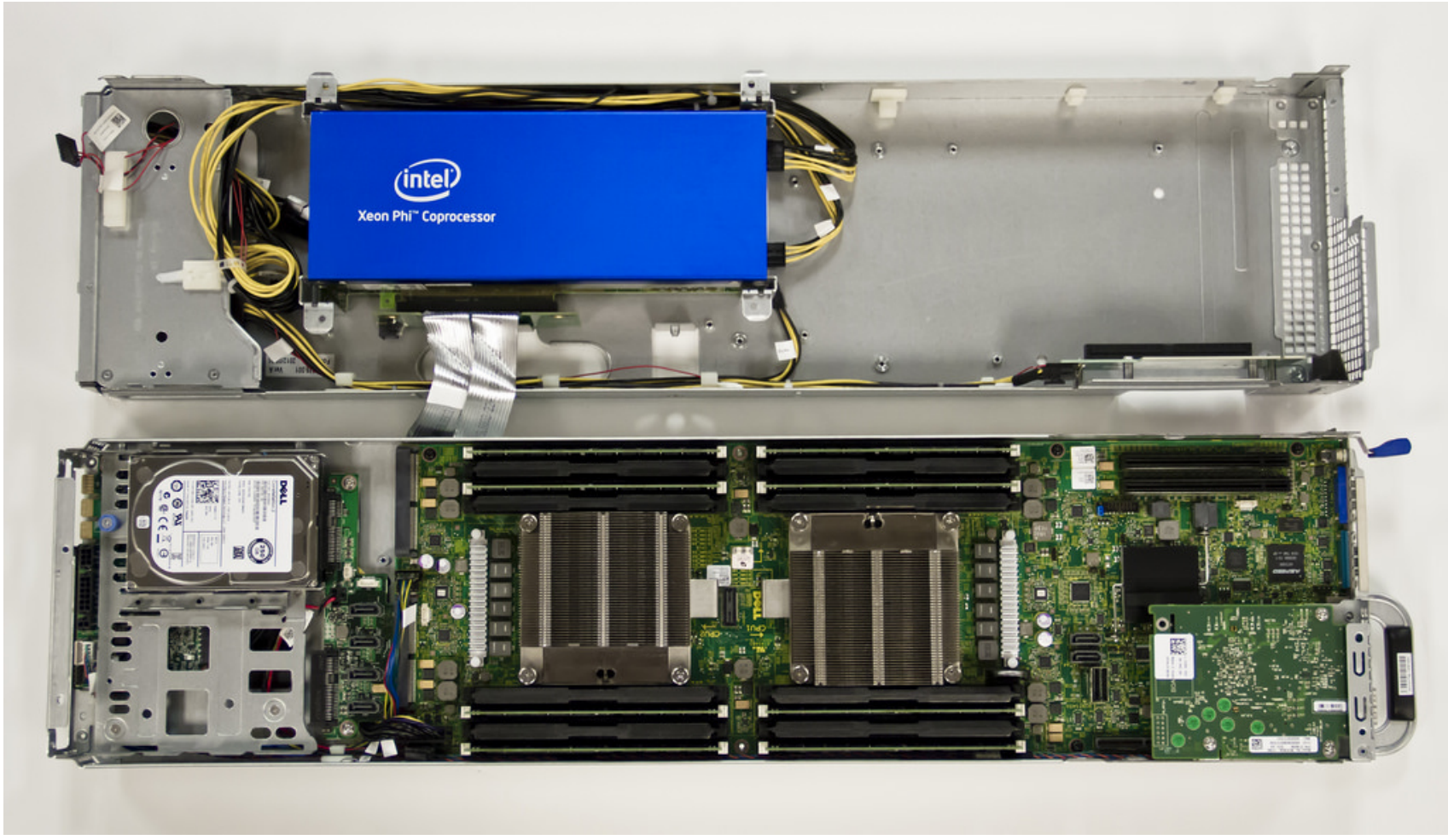
Memory is equally programmable, but not equally accessible

- Different caches, different affinity



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

Picture of NUMA



Interconnects and topologies, theoretical concepts

Topology concepts

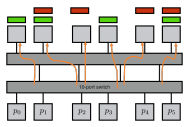
- Hardware characteristics
- Software requirement
- Design: how ‘close’ are processors?

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

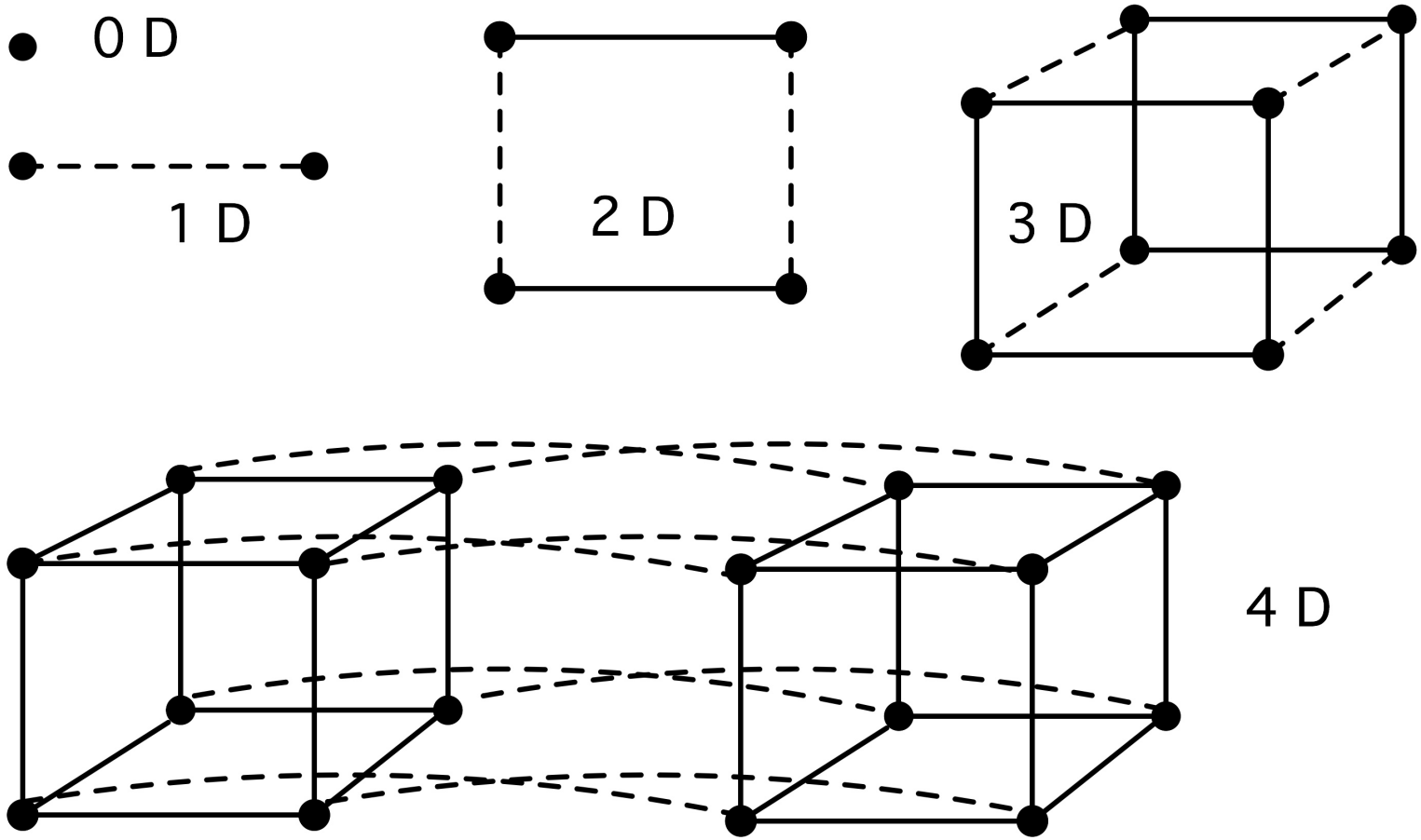
Exercise 5

Broadcast algorithm Flip last bit, flip one before, . . .

Design 3: 2/3-D arrays

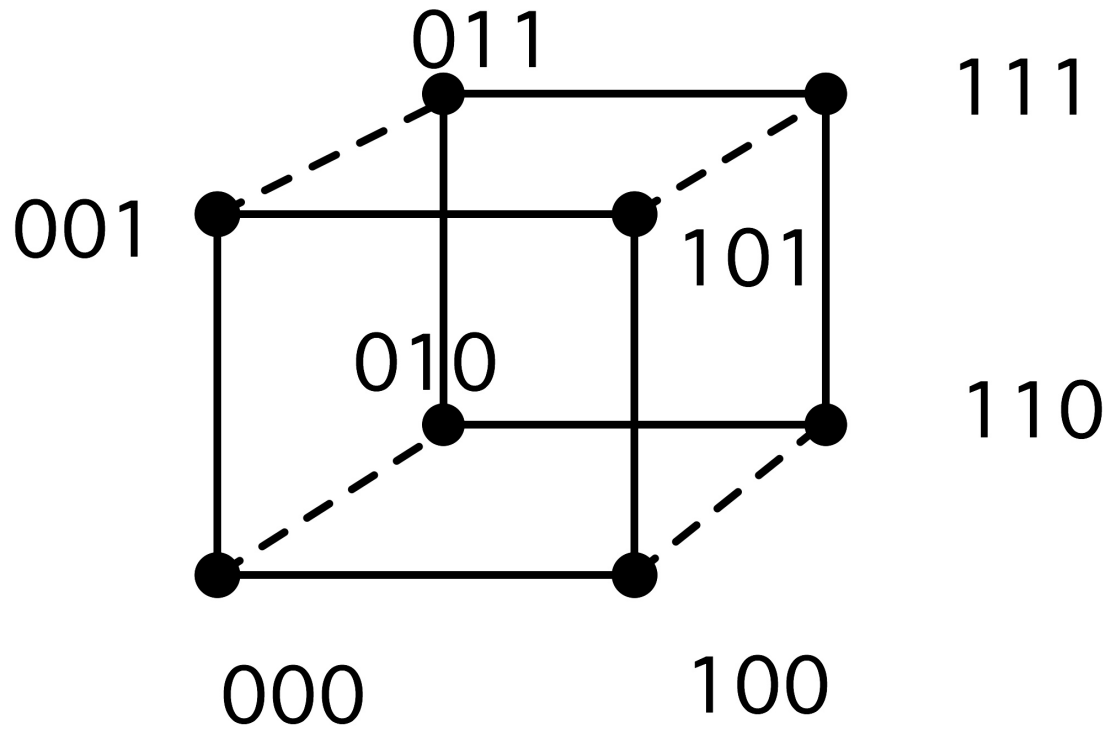
- Degree $2d$, diameter $P^{1/d}$
- Natural design: nature is three-dimensional
- More dimensions: less contention.
  K-machine is 6-dimensional

Design 3: Hypercubes



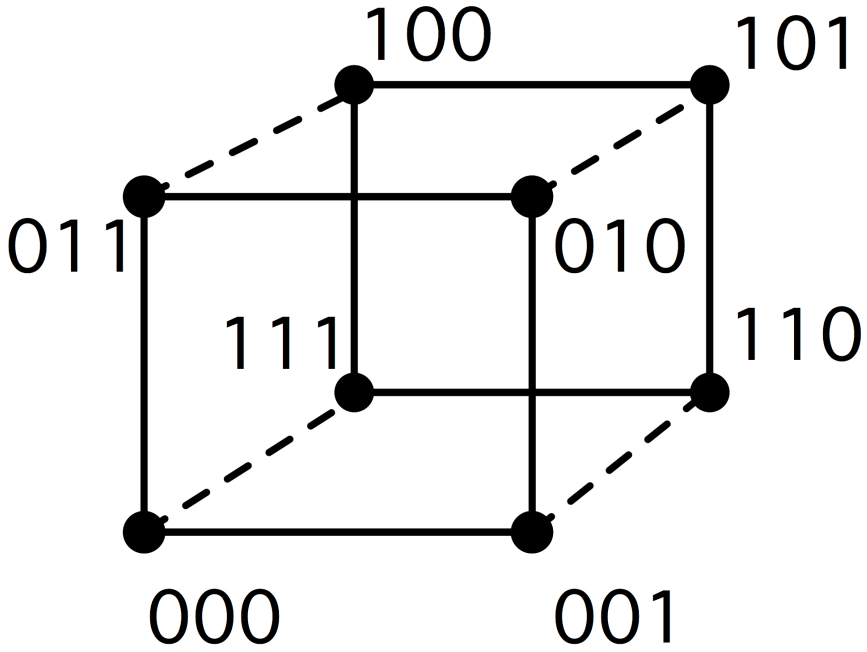
Hypercube numbering

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

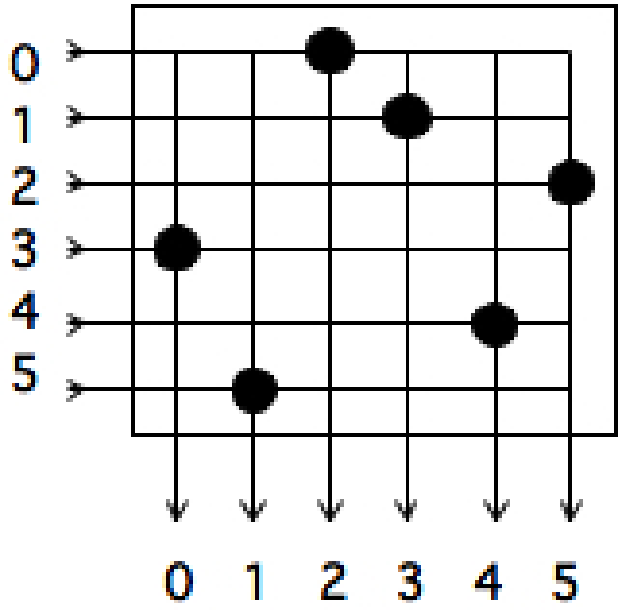
0 0 1 1 : 1 1 0 0

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

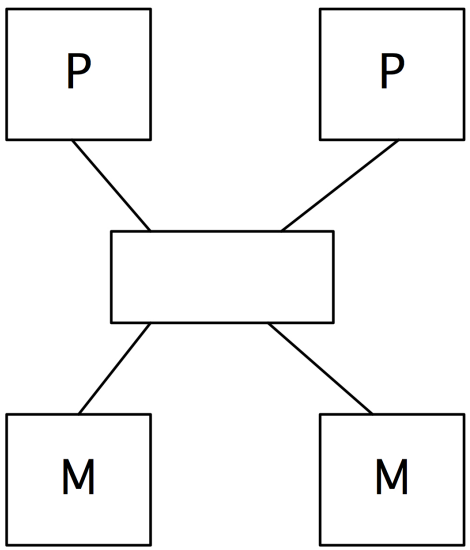


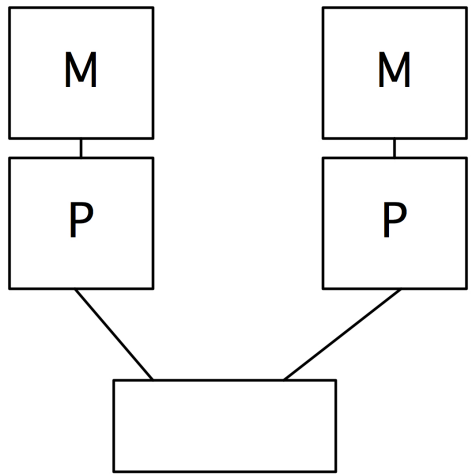
Advantage: non-blocking

Disadvantage: cost

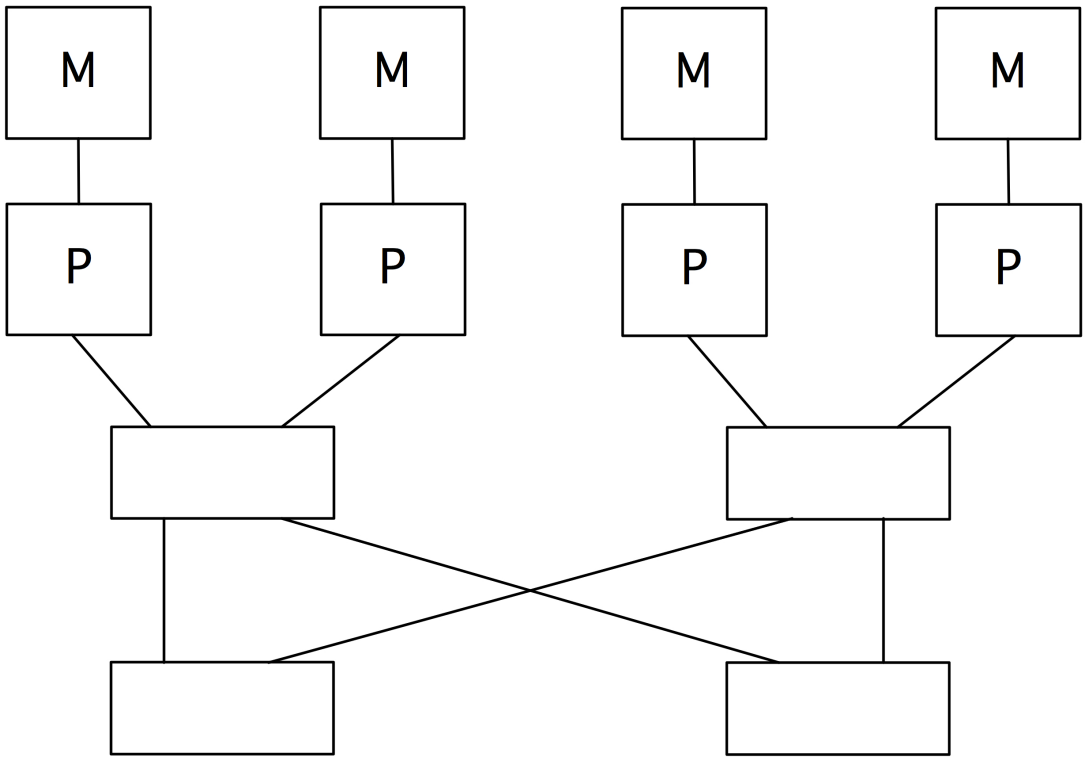
Butterfly exchange

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

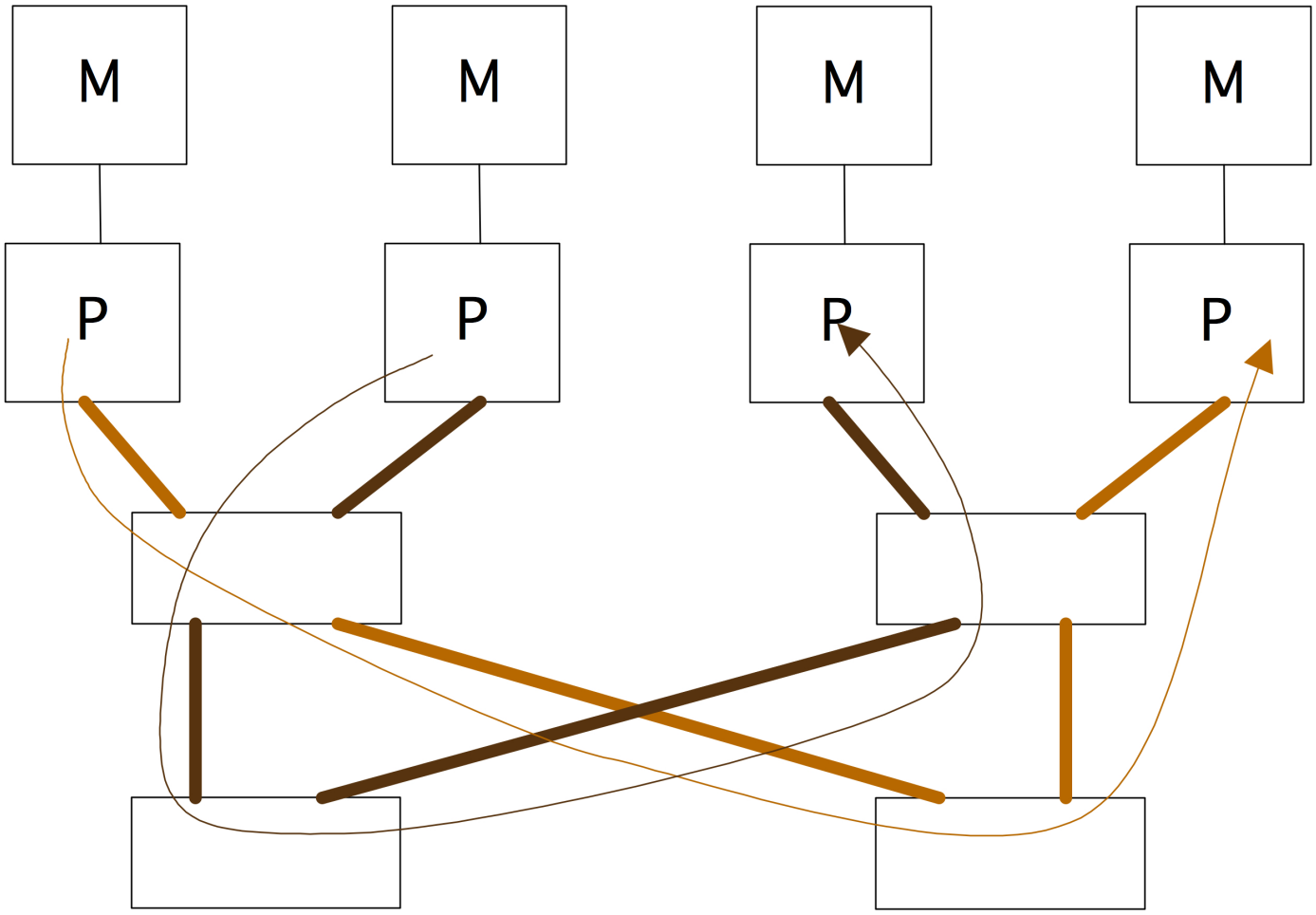




Building up butterflies

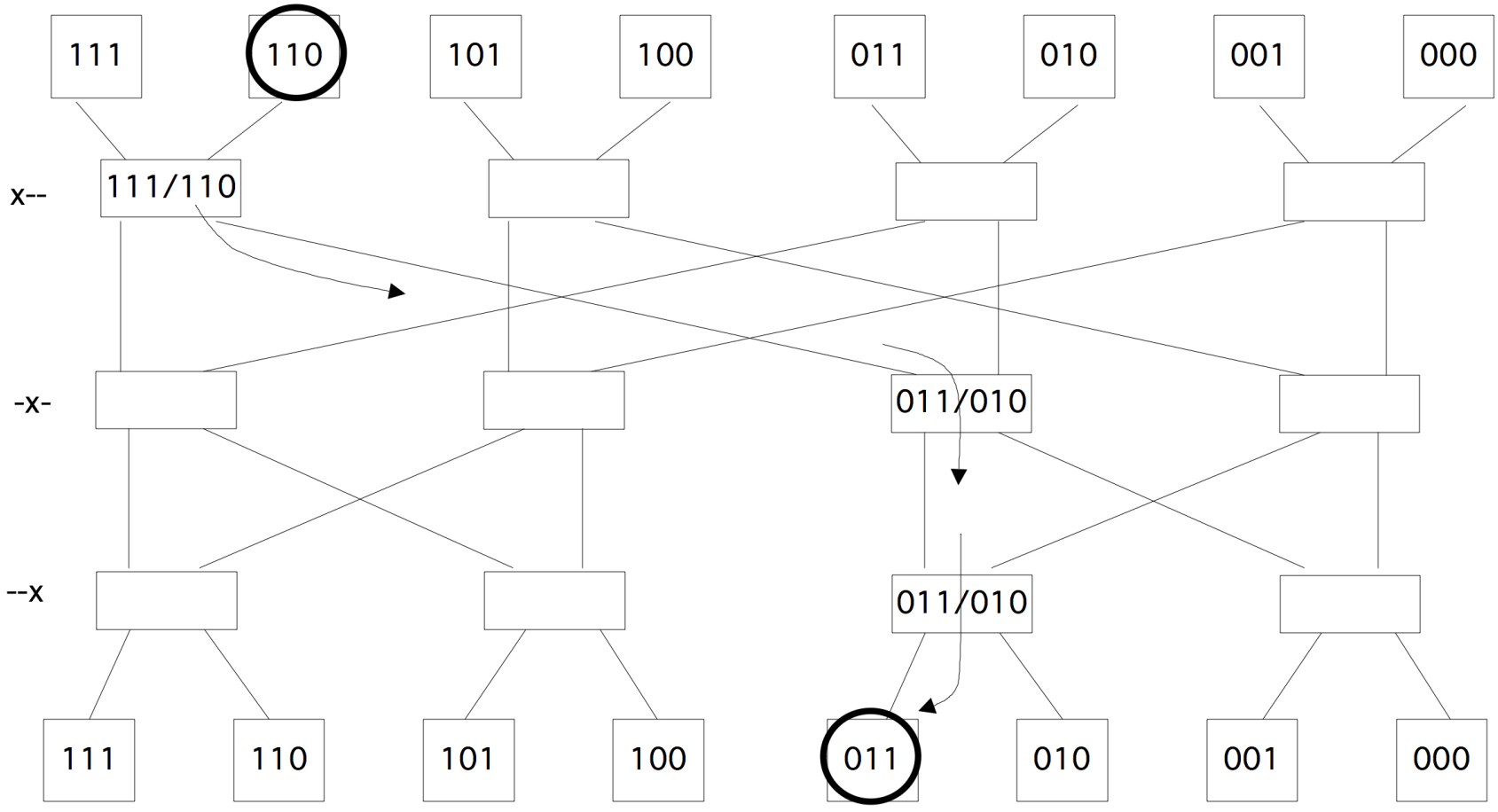


Uniform memory access

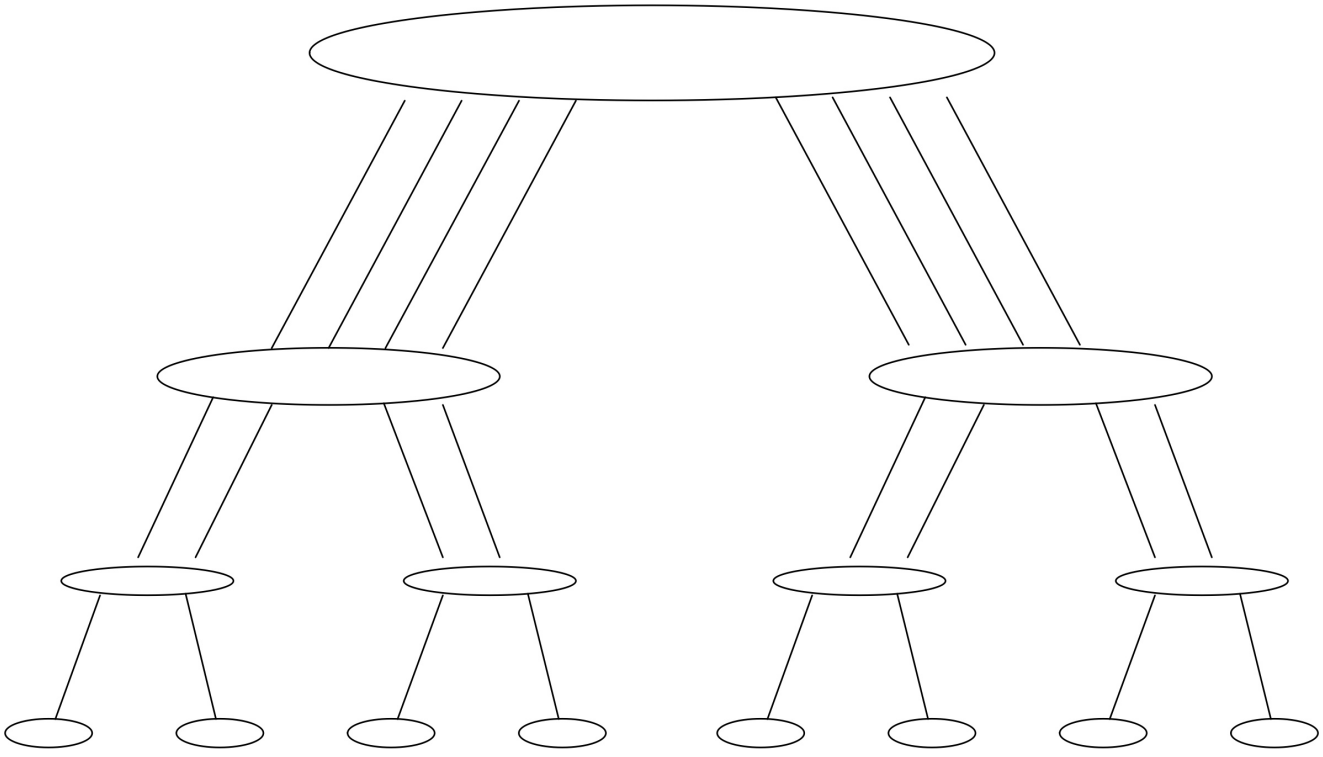


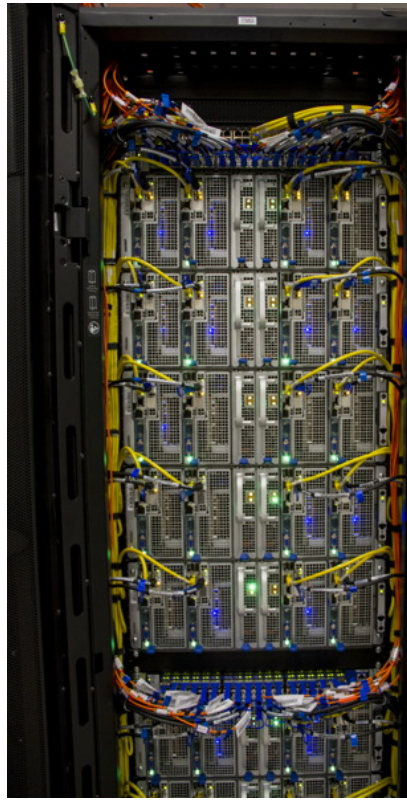
Contention possible

Route calculation

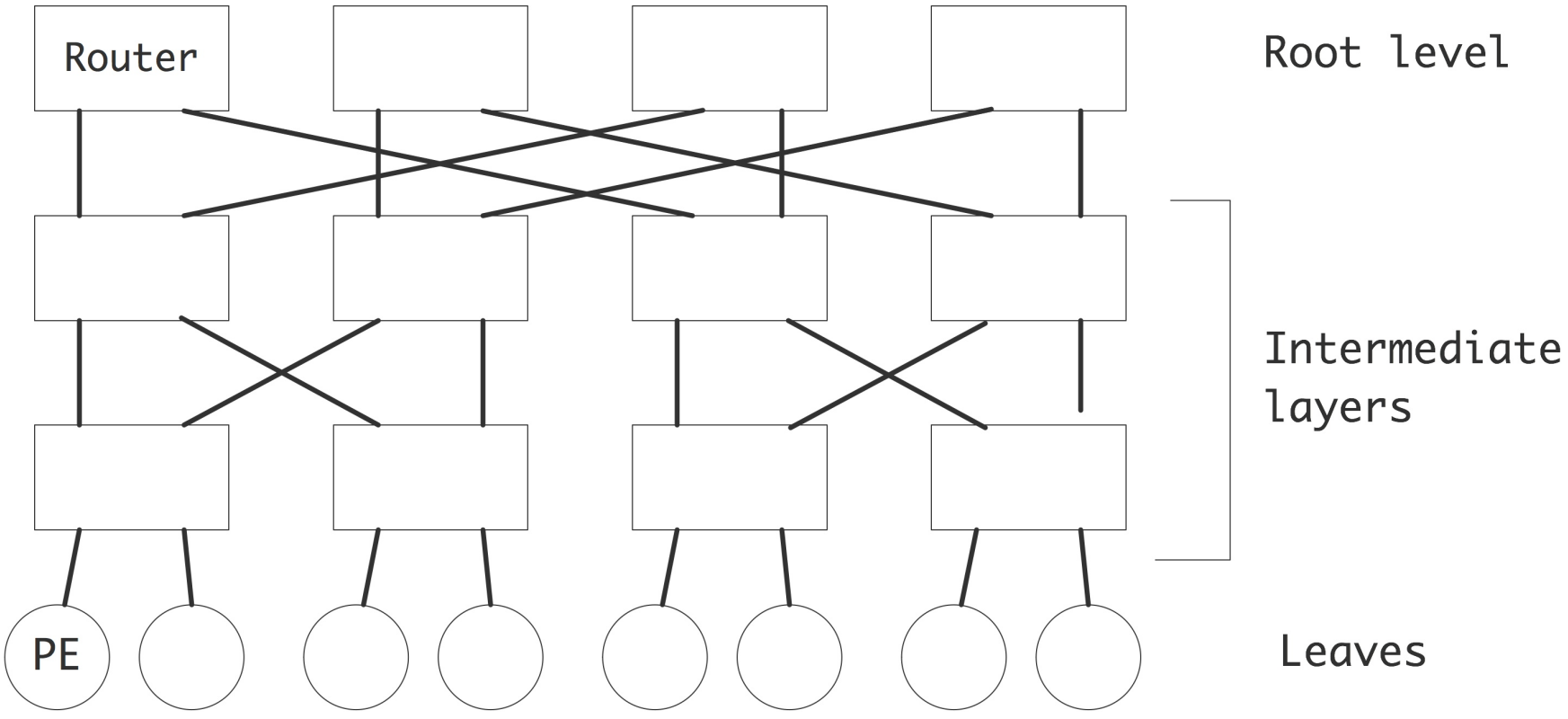


Fat Tree





Fat trees from switching elements

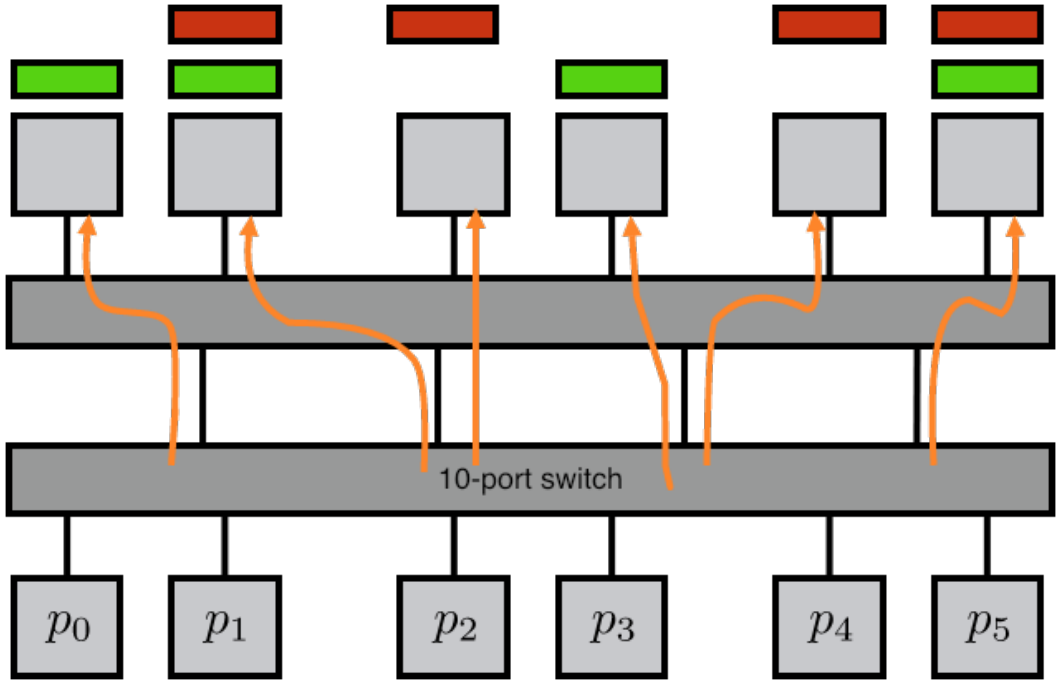


(Clos network)

Fat tree clusters

Exercise 6

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?



Mesh clusters

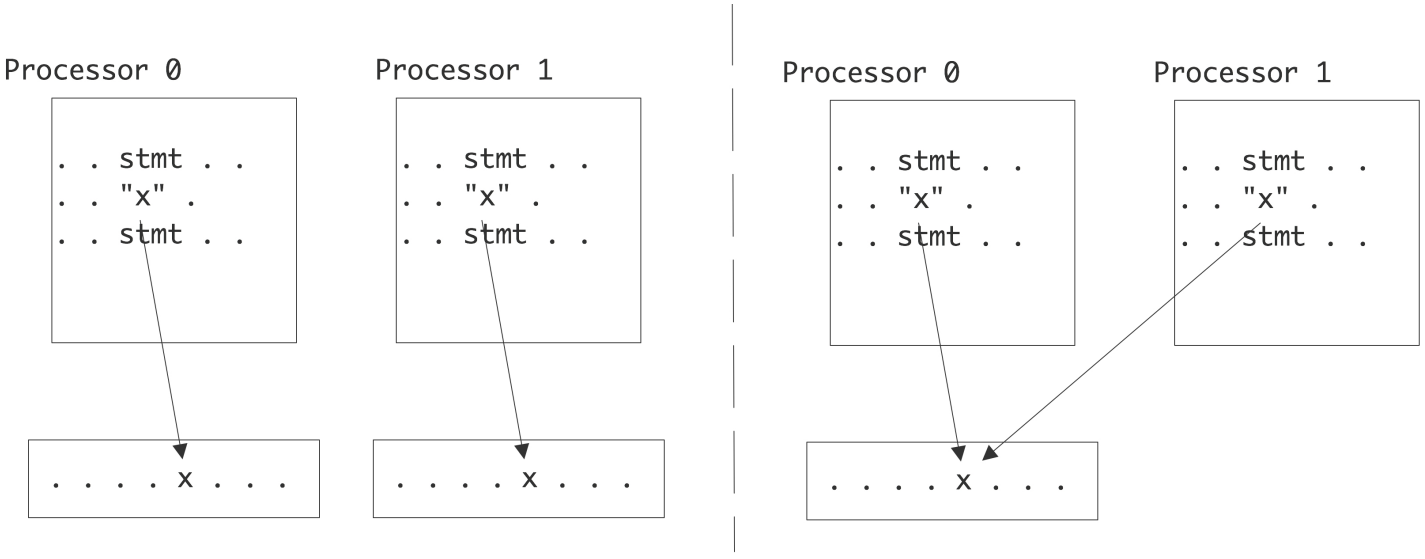
Levels of locality

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

Programming models

Shared vs distributed memory programming

Different memory models:



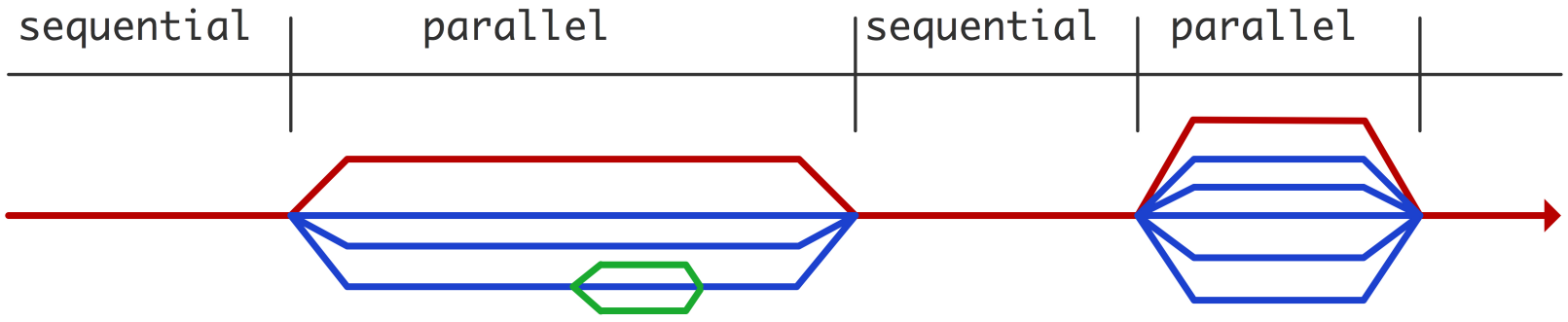
Different questions:

- 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

```
1 pthread_t threads[NTHREADS];
2 printf("forking\n");
3 for (i=0; i<NTHREADS; i++)
4     if (pthread_create(threads+i,NULL,&adder,NULL)!=0)
5         return i+1;
6 printf("joining\n");
7 for (i=0; i<NTHREADS; i++)
8     if (pthread_join(threads[i],NULL)!=0)
9         return NTHREADS+i+1;
```

Race conditions

Init: `I=0`

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 $I = 0$	
set $I = 2$	set $I = 3$	set $I = 2$	set $I = 3$	set $I = 2$	
write $I = 2$			write $I = 3$	write $I = 2$	
	write $I = 3$	write $I = 2$			read $I = 2$
					set $I = 5$
					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

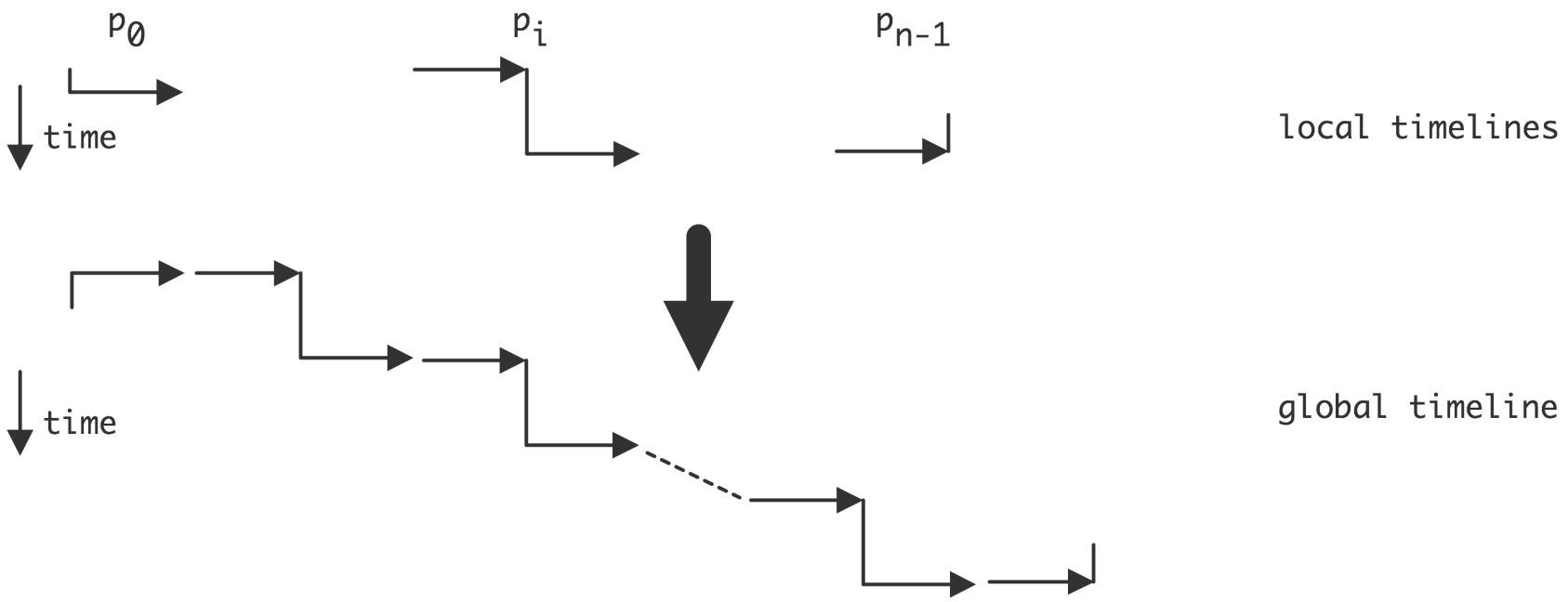
Global vs local view

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

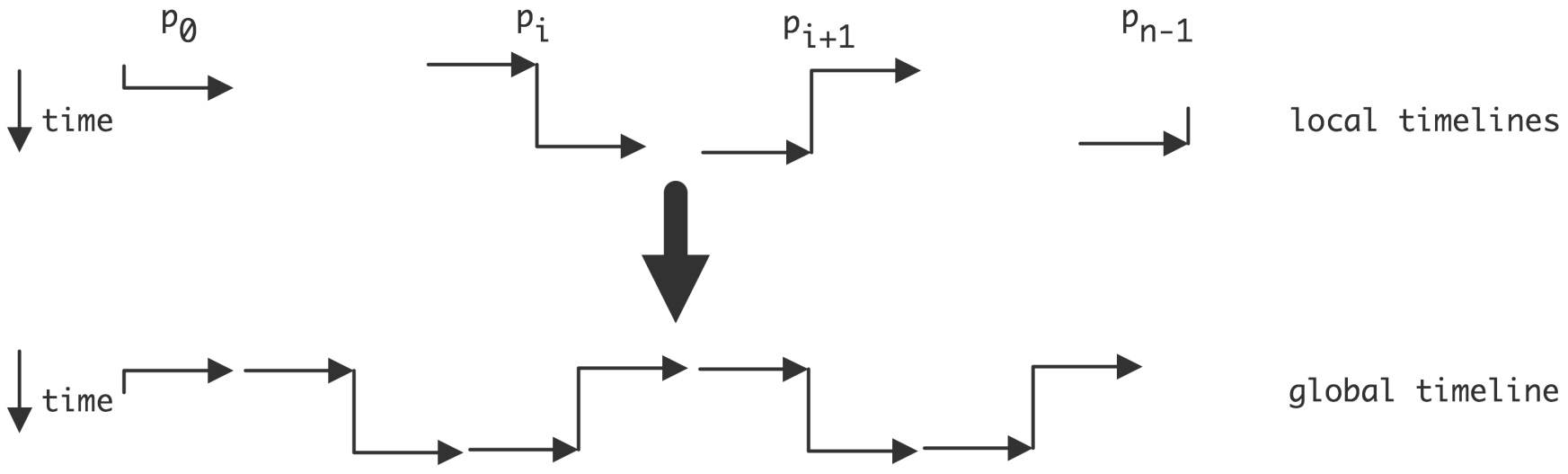
- If I am processor 0 do nothing, otherwise receive a y element from the left, add it to my x element.
- If I am the last processor do nothing, otherwise send my y element to the right.

(Let's think this through. . .)

Global picture



Careful coding



Better approaches

- Non-blocking send/receive
- One-sided

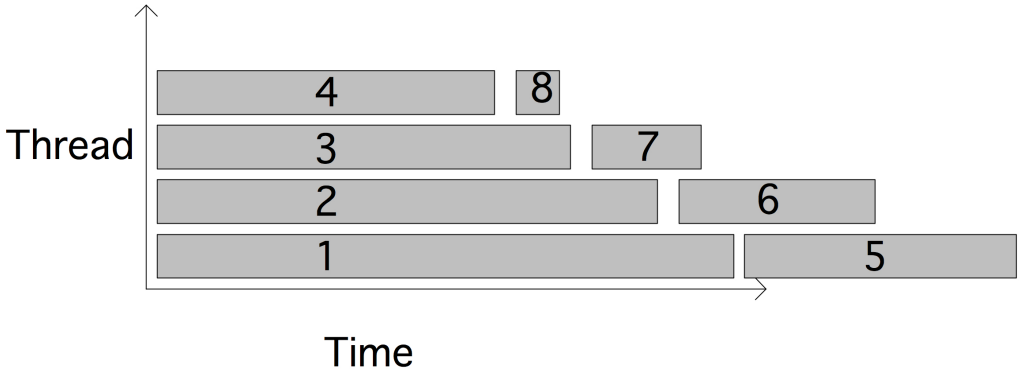
Hybrid/heterogeneous parallelism

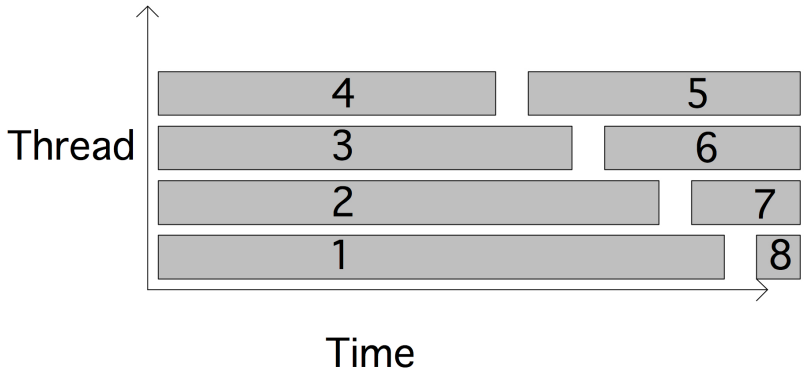
Hybrid computing

- 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:

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

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

Design patterns

Array of Structures

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

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

Operations

Operate

```
1 void shift(node the_point,vector by) {
2   the_point->xcoord += by->xtrans;
3   the_point->ycoord += by->ytrans;
4 }
```

in a loop

```
1 for (i=0; i<n_nodes; i++) {
2   shift(nodes[i],shift_vector);
3 }
```

Along come the 80s

Vector operations

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

and you would iterate

```
1 for (i=0; i<n_nodes; i++) {
2   node_xoords[i] += shift_vector->xtrans;
3   node_yoords[i] += shift_vector->ytrans;
4 }
```

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

```
1  #define N 100*THREADS
2
3  shared int v1[N], v2[N], v1plusv2[N];
4
5  void main()
6  {
7    int i;
8    upc_forall(i=0; i<N; i++; i)
9      v1plusv2[i]=v1[i]+v2[i];
10 }
```

Co-array Fortran example

Explicit dimension for ‘images’:

```
1 Real,dimension(100),codimension[*] :: X
2 Real :: X(100)[*]
3 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++)

Load balancing, locality, space-filling curves

The load balancing problem

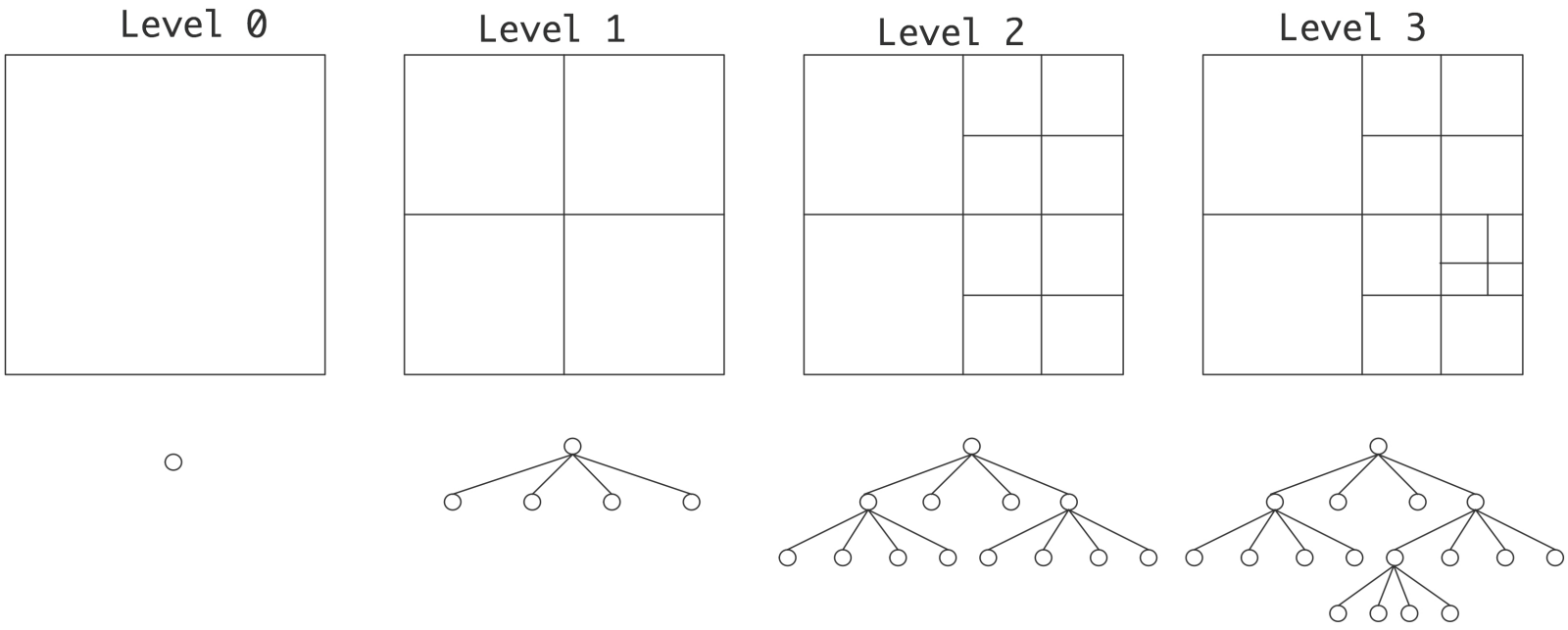
- Application load can change dynamically
  e.g., mesh refinement, time-dependent problems
- Splitting off and merging loads
- No real software support: write application anticipating load management
- Initial balancing: graph partitioners

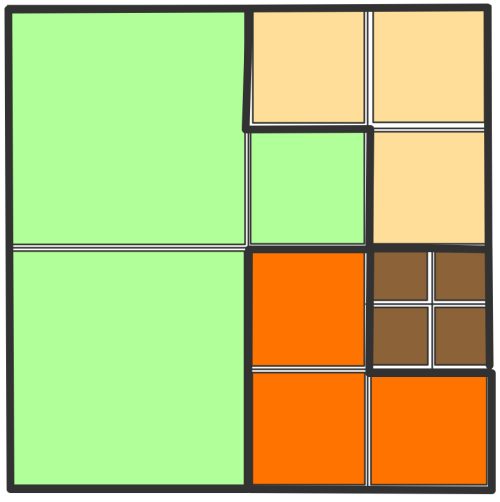
Load balancing and performance

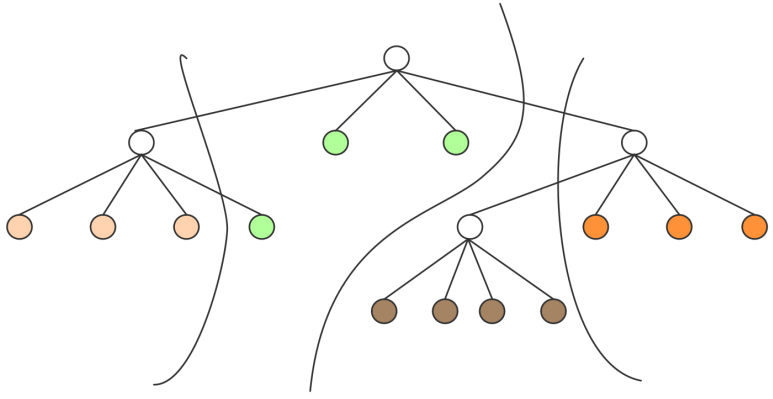
- Assignment to arbitrary processor violates locality
- Need a dynamic load assignment scheme that preserves locality under load migration
- Fairly easy for regular problems, for irregular?

Space-filling curves

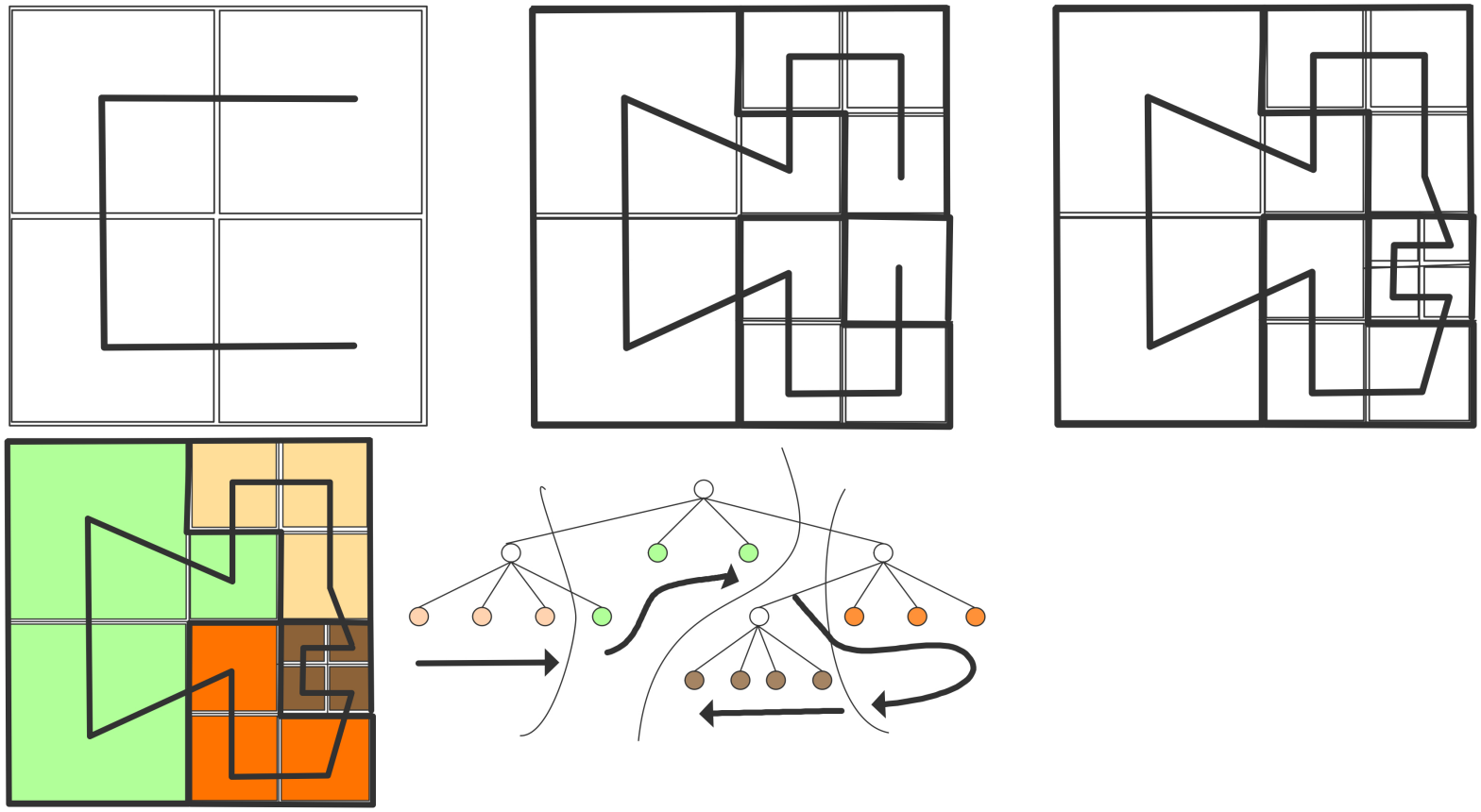
Adaptive refinement and load assignment





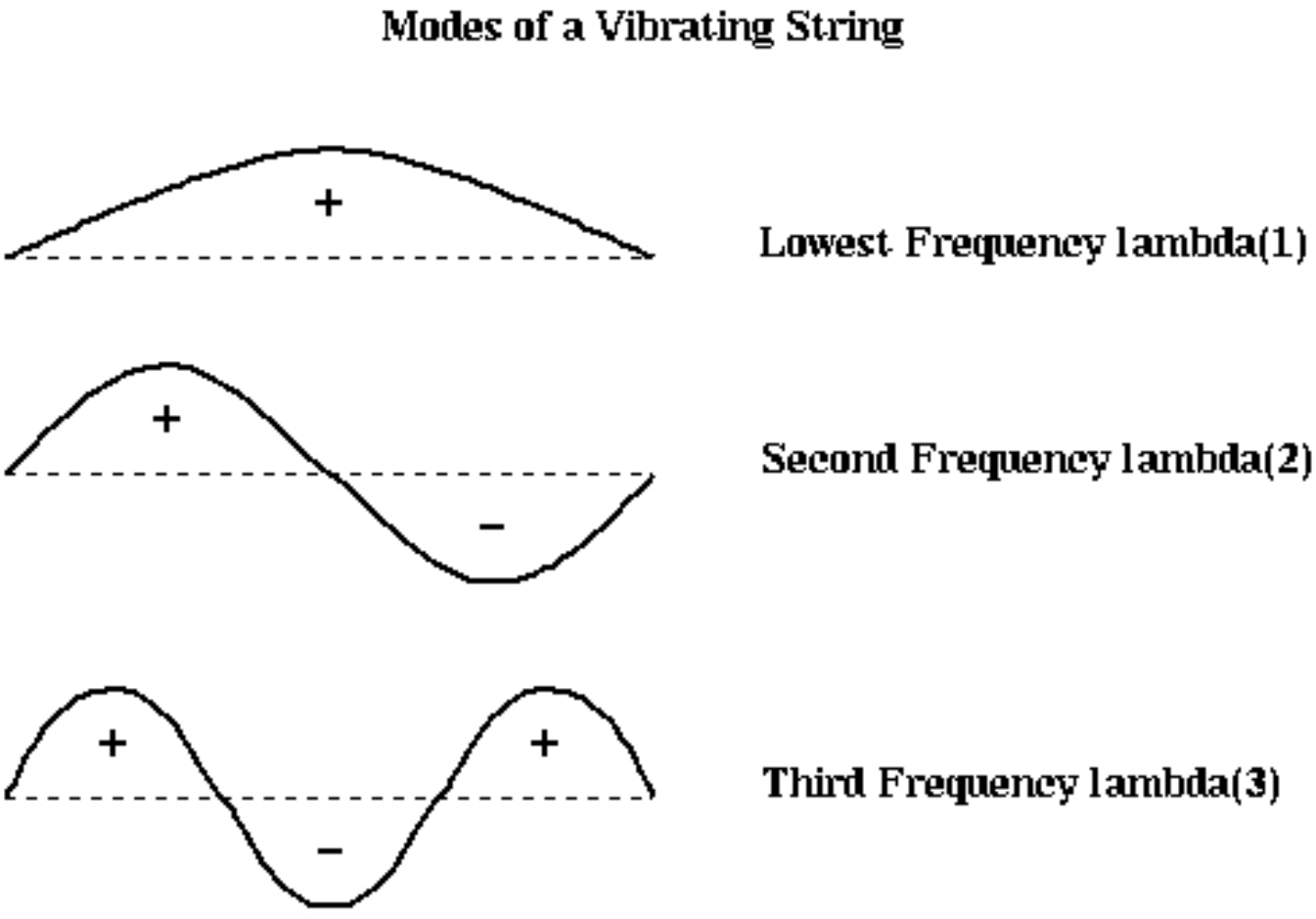


Assignment through Space-Filling Curve



Domain partitioning by Fiedler vectors

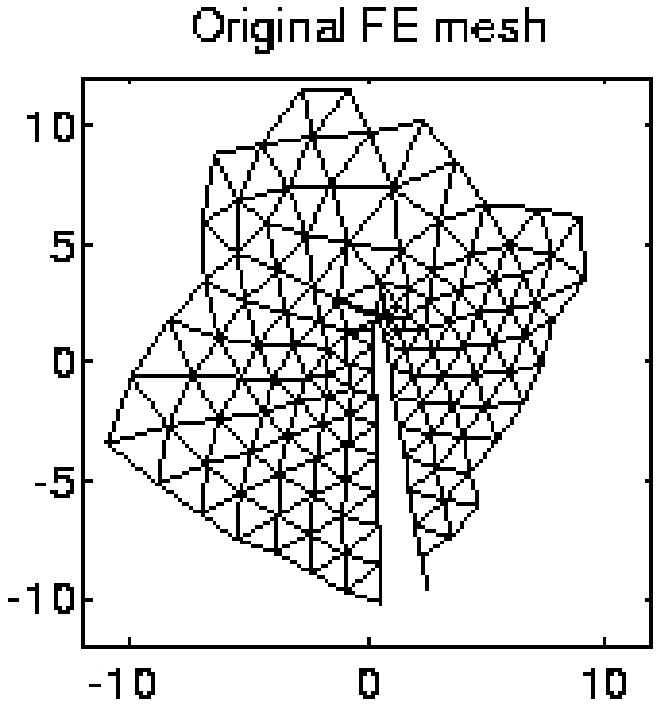
Inspiration from physics

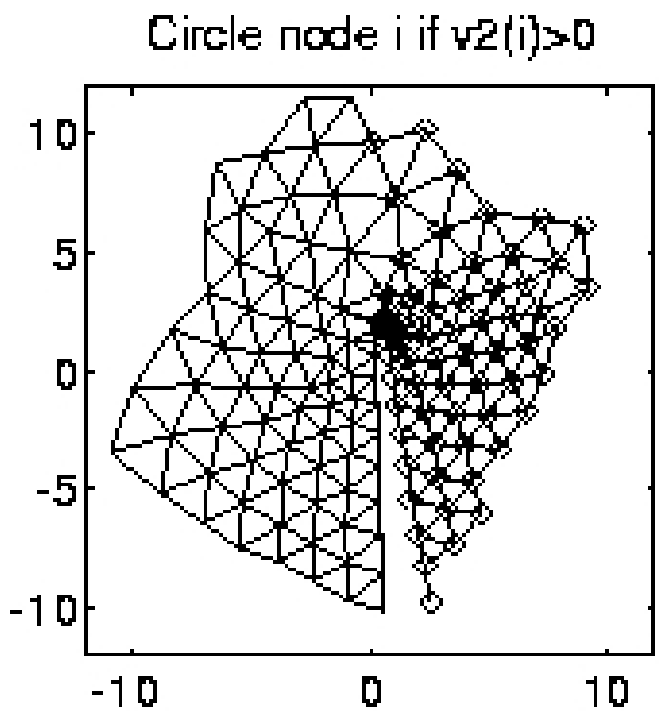


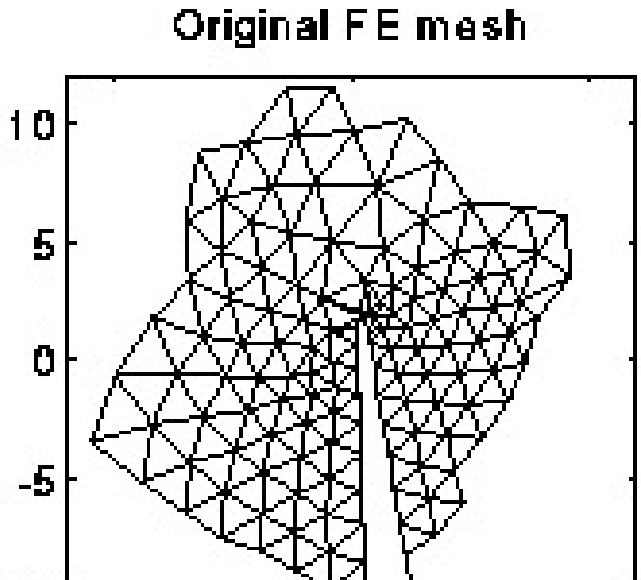
Graph laplacian

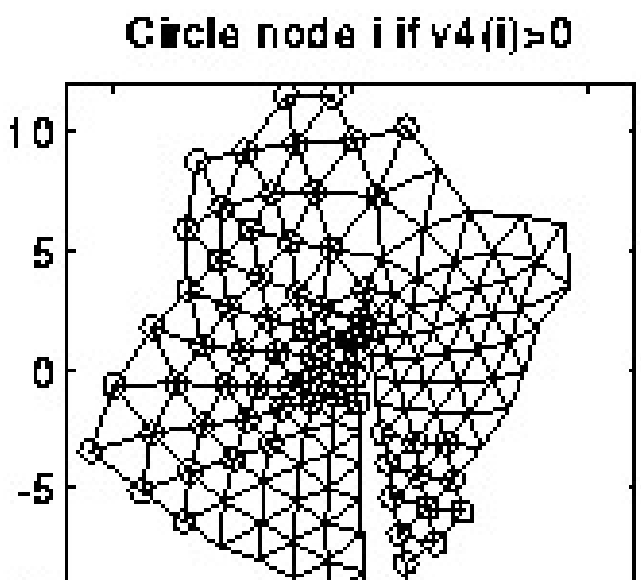
- Set $G_{ij} = -1$ if edge (i, j)
- Set G_{ii} positive to give zero rowsums
- First eigenvector is zero, positive eigenvector
- Second eigenvector has pos/neg, divides in two
- n-th eigenvector divides in n parts

Fiedler in a picture









Collectives

Collectives as building blocks; complexity

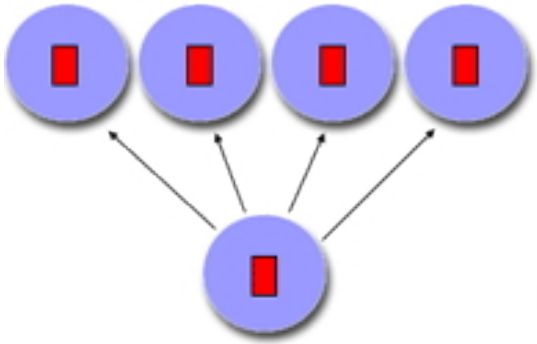
Collectives

Gathering and spreading information:

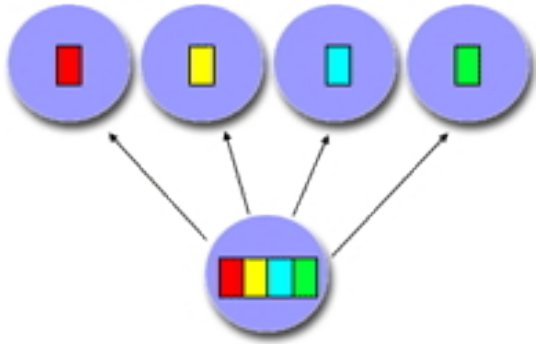
- Every process has data, you want to bring it together;
- One process has data, you want to spread it around.

Root process: the one doing the collecting or disseminating.
Basic cases:

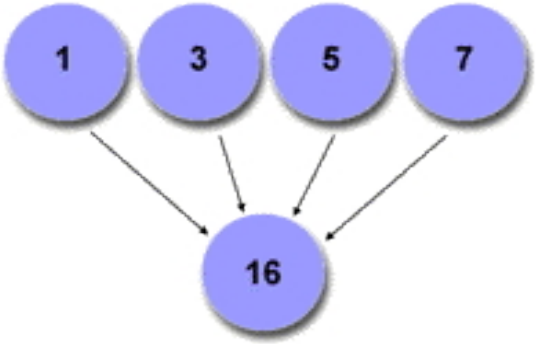
- Collect data: gather.
- Collect data and compute some overall value (sum, max): reduction.
- Send the same data to everyone: broadcast.
- Send individual data to each process: scatter.



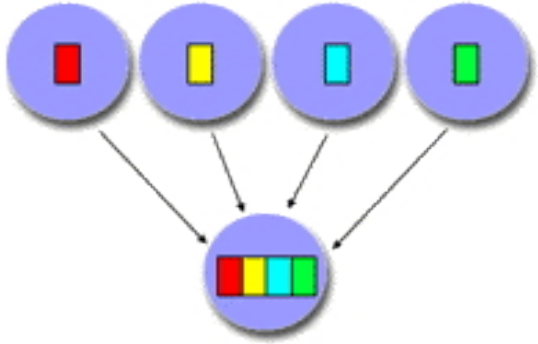
broadcast



scatter



reduction



gather

Collective scenarios

How would you realize the following scenarios with collectives?

- Let each process compute a random number. You want to print the maximum of these numbers to your screen.
- Each process computes a random number again. Now you want to scale these numbers by their maximum.
- Let each process compute a random number. You want to print on what processor the maximum value is computed.

Simple model of parallel computation

- α: message latency
- β: time per word (inverse of bandwidth)
- γ: time per floating point operation

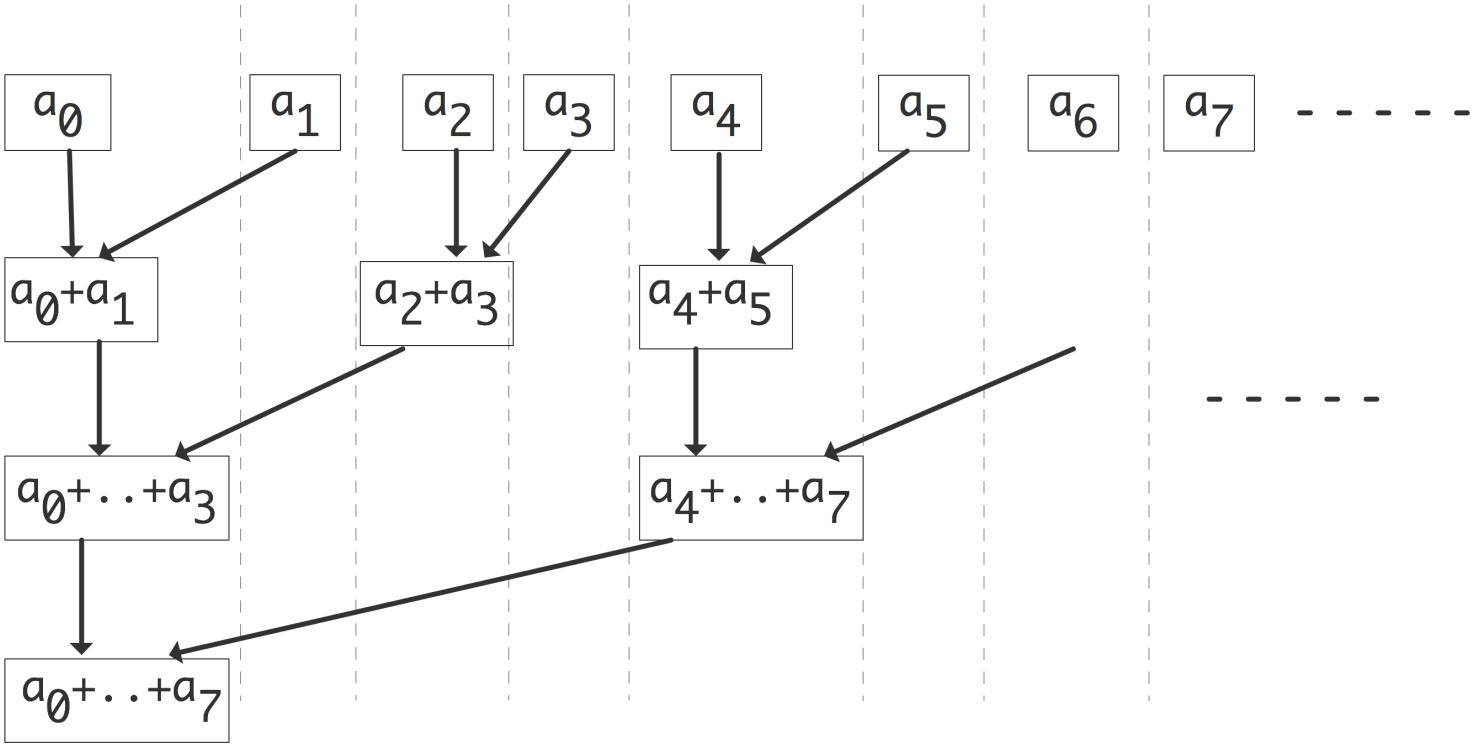
Send n items and do m operations:

$$cost = \alpha + \beta \cdot n + \gamma \cdot m$$

Pure sends: no γ term,
pure computation: no α, β terms,
sometimes mixed: reduction

Model for collectives

- One simultaneous send and receive:
- doubling of active processors
- collectives have a $\alpha \log_2 p$ cost component



Broadcast

	$t=0$	$t=1$	$t=2$
p_0	$x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$	$x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$	x_0, x_1, x_2, x_3
p_1		$x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$	x_0, x_1, x_2, x_3
p_2			x_0, x_1, x_2, x_3
p_3			x_0, x_1, x_2, x_3

On $t=0$, p_0 sends to p_1; on $t=1$ p_0, p_1 send to p_2, p_3.

Optimal complexity:

$$\lceil \log_2 p \rceil \alpha + n\beta.$$

Actual complexity:

$$\lceil \log_2 p \rceil (\alpha + n\beta).$$

Good enough for short vectors.

Long vector broadcast

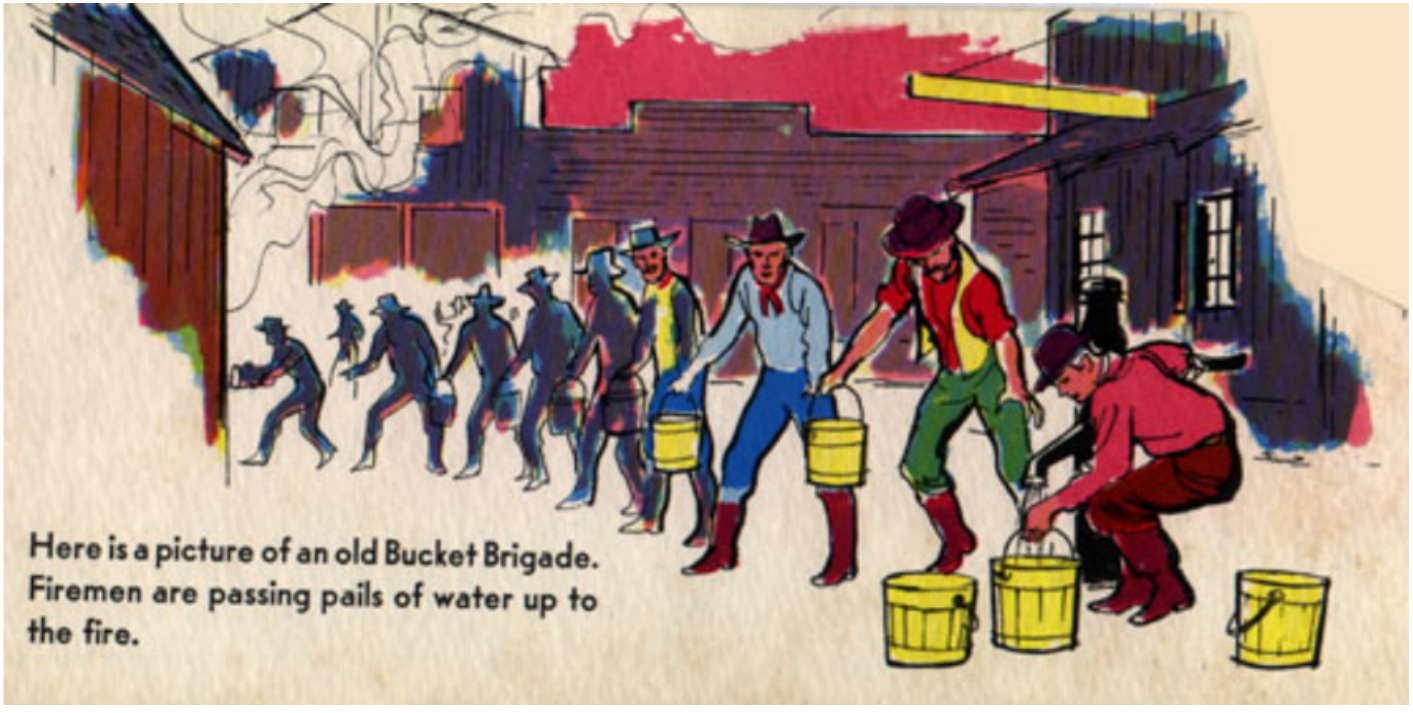
Start with a scatter:

	$t=0$	$t=1$	$t=2$	$t=3$
p_0	$x_0\downarrow, x_1, x_2, x_3$	$x_0, x_1\downarrow, x_2, x_3$	$x_0, x_1, x_2\downarrow, x_3$	$x_0, x_1, x_2, x_3\downarrow$
p_1		x_1		
p_2			x_2	
p_3				x_3

takes $p-1$ messages of size N/p, for a total time of

$$T_{\text{scatter}}(N,P) = (p-1)\alpha + (p-1)\cdot\frac{N}{p}\cdot\beta.$$

Bucket brigade



Long vector broadcast

After the scatter do a bucket-allgather:

	$t = 0$	$t = 1$	*etcetera*
p_0	$x_0 \downarrow$	$x_0 \quad x_3 \downarrow$	$x_0, \quad x_2, x_3$
p_1	$x_1 \downarrow$	$x_0 \downarrow, x_1$	$x_0, x_1, \quad x_3$
p_2	$x_2 \downarrow$	$x_1 \downarrow, x_2$	x_0, x_1, x_2
p_3	$x_3 \downarrow$	$x_2 \downarrow, x_3$	x_1, x_2, x_3

Each partial message gets sent $p-1$ times, so this stage also has a complexity of

$$T_{\text{bucket}}(N,P) = (p-1)\alpha + (p-1)\cdot\frac{N}{p}\cdot\beta.$$

Better if N large.

Reduce

Optimal complexity:

$$\lceil \log_2 p \rceil \alpha + n\beta + \frac{p-1}{p}\gamma n.$$

Spanning tree algorithm:

	$t=1$	$t=2$	$t=3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, x_3^{(0)}$	$x_0^{(0:1)}, x_1^{(0:1)}, x_2^{(0:1)}, x_3^{(0:1)}$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x$
p_1	$x_0^{(1)} \uparrow, x_1^{(1)} \uparrow, x_2^{(1)} \uparrow, x_3^{(1)} \uparrow$		
p_2	$x_0^{(2)}, x_1^{(2)}, x_2^{(2)}, x_3^{(2)}$	$x_0^{(2:3)} \uparrow, x_1^{(2:3)} \uparrow, x_2^{(2:3)} \uparrow, x_3^{(2:3)} \uparrow$	
p_3	$x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)} \uparrow, x_3^{(3)} \uparrow$		

Running time

$$\lceil \log_2 p \rceil (\alpha + n\beta + \frac{p-1}{p}\gamma n).$$

Good enough for short vectors.

Allreduce

Allreduce $\equiv$ Reduce+Broadcast

	$t=1$	$t=2$	$t=3$
p_0	$x_0^{(0)}\downarrow, x_1^{(0)}\downarrow, x_2^{(0)}\downarrow, x_3^{(0)}\downarrow$	$x_0^{(0:1)}\downarrow\downarrow, x_1^{(0:1)}\downarrow\downarrow, x_2^{(0:1)}\downarrow\downarrow, x_3^{(0:1)}\downarrow\downarrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_1	$x_0^{(1)}\uparrow, x_1^{(1)}\uparrow, x_2^{(1)}\uparrow, x_3^{(1)}\uparrow$	$x_0^{(0:1)}\downarrow\downarrow, x_1^{(0:1)}\downarrow\downarrow, x_2^{(0:1)}\downarrow\downarrow, x_3^{(0:1)}\downarrow\downarrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_2	$x_0^{(2)}\downarrow, x_1^{(2)}\downarrow, x_2^{(2)}\downarrow, x_3^{(2)}\downarrow$	$x_0^{(2:3)}\uparrow\uparrow, x_1^{(2:3)}\uparrow\uparrow, x_2^{(2:3)}\uparrow\uparrow, x_3^{(2:3)}\uparrow\uparrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_3	$x_0^{(3)}\uparrow, x_1^{(3)}\uparrow, x_2^{(3)}\uparrow, x_3^{(3)}\uparrow$	$x_0^{(2:3)}\uparrow\uparrow, x_1^{(2:3)}\uparrow\uparrow, x_2^{(2:3)}\uparrow\uparrow, x_3^{(2:3)}\uparrow\uparrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$

Same running time as regular reduce!

Allgather

Gather n elements: each processor owns n/p;
optimal running time

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n\beta.$$

	$t=1$	$t=2$	$t=3$
p_0	$x_0 \downarrow$	$x_0 x_1 \downarrow$	$x_0 x_1 x_2 x_3$
p_1	$x_1 \uparrow$	$x_0 x_1 \downarrow$	$x_0 x_1 x_2 x_3$
p_2	$x_2 \downarrow$	$x_2 x_3 \uparrow$	$x_0 x_1 x_2 x_3$
p_3	$x_3 \uparrow$	$x_2 x_3 \uparrow$	$x_0 x_1 x_2 x_3$

Same time as gather, half of gather-and-broadcast.

Reduce-scatter

	$t=1$	$t=2$	$t=3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)} \downarrow, x_3^{(0)} \downarrow$	$x_0^{(0:2:2)}, x_1^{(0:2:2)} \downarrow$	$x_0^{(0:3)}$
p_1	$x_0^{(1)}, x_1^{(1)}, x_2^{(1)} \downarrow, x_3^{(1)} \downarrow$	$x_0^{(1:3:2)} \uparrow, x_1^{(1:3:2)}$	$x_1^{(0:3)}$
p_2	$x_0^{(2)} \uparrow, x_1^{(2)} \uparrow, x_2^{(2)}, x_3^{(2)}$	$x_2^{(0:2:2)}, x_3^{(0:2:2)} \downarrow$	$x_2^{(0:3)}$
p_3	$x_0^{(3)} \uparrow, x_1^{(3)} \uparrow, x_2^{(3)}, x_3^{(3)}$	$x_0^{(1:3:2)} \uparrow, x_1^{(1:3:2)}$	$x_3^{(0:3)}$

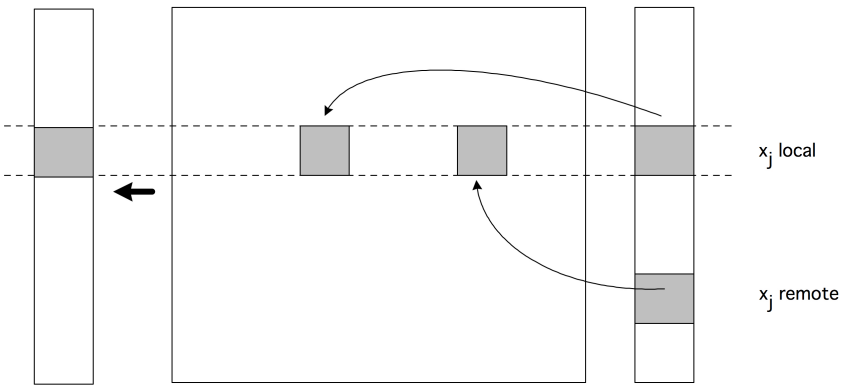
$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n(\beta + \gamma).$$

Applications

Scalability analysis of dense matrix-vector product

Parallel matrix-vector product; general

- Assume a division by block rows
- Every processor p has a set of row indices I_p

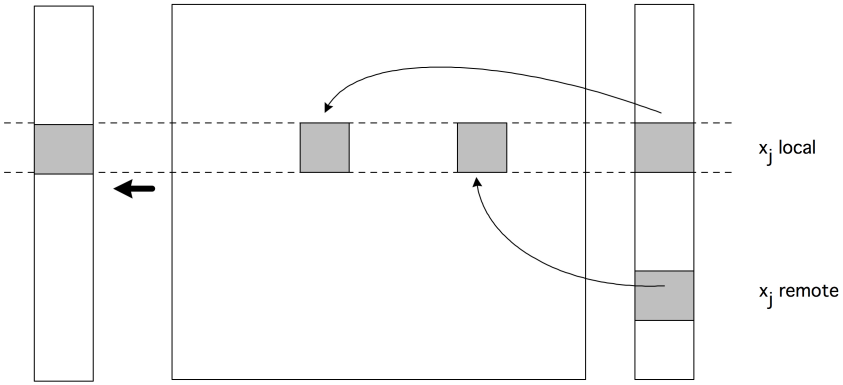


Mvp on processor p:

$$\forall_{i \in I_p} : y_i = \sum_j a_{ij} x_j = \sum_q \sum_{j \in I_q} a_{ij} x_j$$

Local and remote operations

Local and remote parts:



$$\forall_{i \in I_p} : y_i = \sum_{j \in I_p} a_{ij} x_j + \sum_{q \neq p} \sum_{j \in I_q} a_{ij} x_j$$

Local part I_p can be executed right away, I_q requires communication.

How to deal with remote parts

- Very flexible: mix of working on local parts, and receiving remote parts.
- More orchestrated:
  1. each process gets a full copy of the input vector (how?)
  2. then operates on the whole input

  Compare?

(Are we making a big assumption here?)

Dense MVP

- Separate communication and computation:
- first allgather
- then matrix-vector product

Cost computation 1.

Algorithm:

Step	Cost (lower bound)
Allgather x_i so that x is available on all nodes	
Locally compute $y_i = A_i x$	$\approx 2\frac{n^2}{P}\gamma$

Allgather

Assume that data arrives over a binary tree:

- latency $\alpha \log_2 P$
- transmission time, receiving n/P elements from $P-1$ processors

Algorithm with cost:

Step	Cost (lower bound)
Allgather x_i so that x is available on all nodes	$\lceil \log_2(P) \rceil \alpha + \frac{P-1}{P} n\beta \approx \log_2(P)\alpha + n\beta$
Locally compute $y_i = A_i x$	$\approx 2\frac{n^2}{P}\gamma$

Parallel efficiency

Speedup:

$$\begin{aligned} S_p^{\text{1D-row}}(n) &= \frac{T_1(n)}{T_p^{\text{1D-row}}(n)} \\ &= \frac{2n^2\gamma}{2\frac{n^2}{p}\gamma + \log_2(p)\alpha + n\beta} \\ &= \frac{p}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} \end{aligned}$$

Efficiency:

$$\begin{aligned} E_p^{\text{1D-row}}(n) &= \frac{S_p^{\text{1D-row}}(n)}{p} \\ &= \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}. \end{aligned}$$

Strong scaling, weak scaling?

Optimistic scaling

Processors fixed, problem grows:

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}.$$

Roughly $E_p \sim 1 - n^{-1}$

Strong scaling

Problem fixed, $p \to \infty$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Strong scaling

Problem fixed, $p \to \infty$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p \log_2(p)}{2n^2} \frac{\alpha}{\gamma} + \frac{p}{2n} \frac{\beta}{\gamma}}.$$

Roughly $E_p \sim p^{-1}$

Weak scaling

Memory fixed:

$$M = n^2/p$$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2\sqrt{M}}\frac{\beta}{\gamma}}$$

Weak scaling

Memory fixed:

$$M = n^2/p$$

$$E_p^{\text{1D-row}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2\sqrt{M}}\frac{\beta}{\gamma}}$$

Does not scale: $E_p \sim 1/\sqrt{p}$

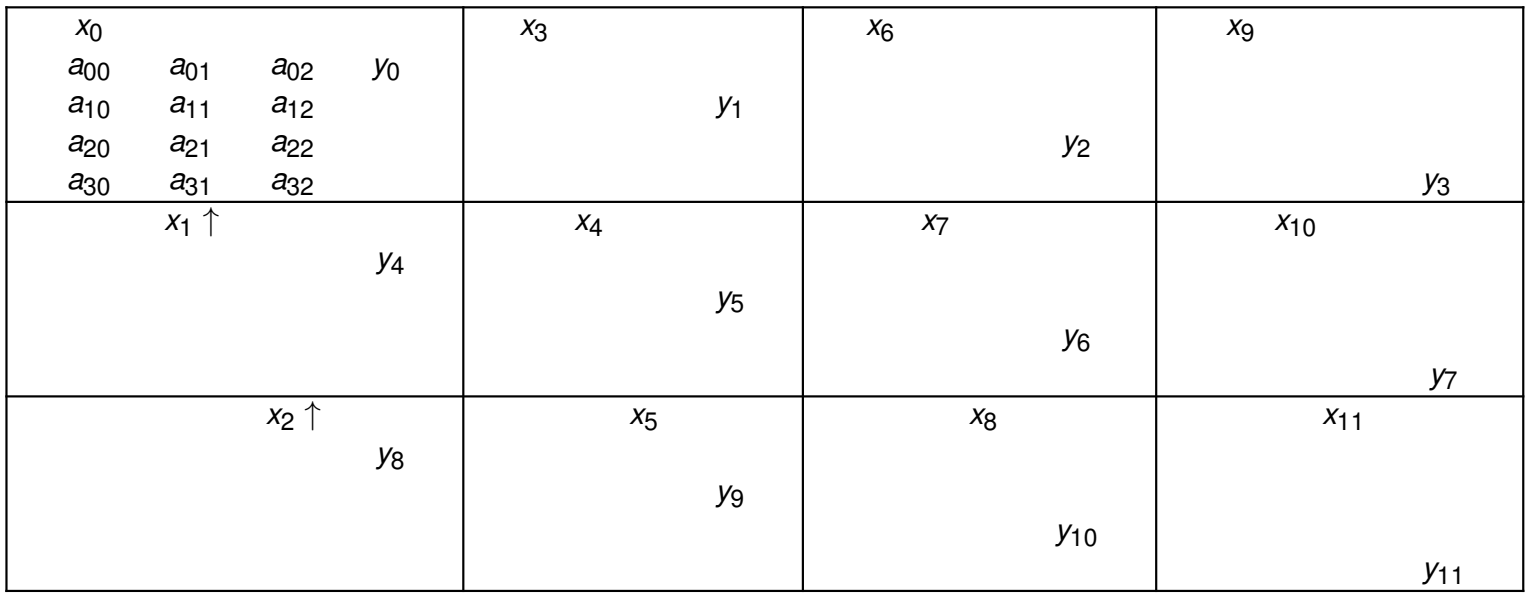
problem in β term: too much communication

Two-dimensional partitioning

x_0				x_3				x_6				x_9			
a_{00}	a_{01}	a_{02}	y_0	a_{03}	a_{04}	a_{05}		a_{06}	a_{07}	a_{08}		a_{09}	$a_{0,10}$	$a_{0,11}$	
a_{10}	a_{11}	a_{12}		a_{13}	a_{14}	a_{15}	y_1	a_{16}	a_{17}	a_{18}		a_{19}	$a_{1,10}$	$a_{1,11}$	
a_{20}	a_{21}	a_{22}		a_{23}	a_{24}	a_{25}		a_{26}	a_{27}	a_{28}	y_2	a_{29}	$a_{2,10}$	$a_{2,11}$	
a_{30}	a_{31}	a_{32}		a_{33}	a_{34}	a_{35}		a_{37}	a_{37}	a_{38}		a_{39}	$a_{3,10}$	$a_{3,11}$	
	x_1				x_4				x_7				x_{10}		
a_{40}	a_{41}	a_{42}	y_4	a_{43}	a_{44}	a_{45}		a_{46}	a_{47}	a_{48}		a_{49}	$a_{4,10}$	$a_{4,11}$	
a_{50}	a_{51}	a_{52}		a_{53}	a_{54}	a_{55}	y_5	a_{56}	a_{57}	a_{58}		a_{59}	$a_{5,10}$	$a_{5,11}$	
a_{60}	a_{61}	a_{62}		a_{63}	a_{64}	a_{65}		a_{66}	a_{67}	a_{68}	y_6	a_{69}	$a_{6,10}$	$a_{6,11}$	
a_{70}	a_{71}	a_{72}		a_{73}	a_{74}	a_{75}		a_{77}	a_{77}	a_{78}		a_{79}	$a_{7,10}$	$a_{7,11}$	
		x_2				x_5				x_8					x_{11}
a_{80}	a_{81}	a_{82}	y_8	a_{83}	a_{84}	a_{85}		a_{86}	a_{87}	a_{88}		a_{89}	$a_{8,10}$	$a_{8,11}$	
a_{90}	a_{91}	a_{92}		a_{93}	a_{94}	a_{95}	y_9	a_{96}	a_{97}	a_{98}		a_{99}	$a_{9,10}$	$a_{9,11}$	
$a_{10,0}$	$a_{10,1}$	$a_{10,2}$		$a_{10,3}$	$a_{10,4}$	$a_{10,5}$		$a_{10,6}$	$a_{10,7}$	$a_{10,8}$	y_{10}	$a_{10,9}$	$a_{10,10}$	$a_{10,11}$	
$a_{11,0}$	$a_{11,1}$	$a_{11,2}$		$a_{11,3}$	$a_{11,4}$	$a_{11,5}$		$a_{11,7}$	$a_{11,7}$	$a_{11,8}$		$a_{11,9}$	$a_{11,10}$	$a_{11,11}$	

Two-dimensional partitioning

Processor grid $p = r \times c$, assume $r, c \approx \sqrt{p}$.



Key to the algorithm

- Consider block (i, j)
- it needs to multiply by the xs in column j
- it produces part of the result of row i

Algorithm

- Collecting x_j on each processor p_{ij} by an *allgather* inside the processor columns.
- Each processor p_{ij} then computes $y_{ij} = A_{ij}x_j$.
- Gathering together the pieces y_{ij} in each processor row to form y_i, distribute this over the processor row: combine to form a *reduce-scatter*.
- Setup for the next A or A^t product

Analysis 1.

Step	Cost (lower bound)
Allgather x_i's within columns	$\lceil \log_2(r) \rceil \alpha + \frac{r-1}{p} n\beta$ $\approx \log_2(r)\alpha + \frac{n}{c}\beta$
Perform local matrix-vector multiply	$\approx 2\frac{n^2}{p}\gamma$
Reduce-scatter y_i's within rows	

Reduce-scatter

Time:

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n(\beta + \gamma).$$

Step	Cost (lower bound)
Allgather x_i's within columns	$\lceil \log_2(r) \rceil \alpha + \frac{r-1}{p} n\beta$ $\approx \log_2(r)\alpha + \frac{n}{c}\beta$
Perform local matrix-vector multiply	$\approx 2\frac{n^2}{p}\gamma$
Reduce-scatter y_i's within rows	$\lceil \log_2(c) \rceil \alpha + \frac{c-1}{p} n\beta + \frac{c-1}{p} n\gamma$ $\approx \log_2(c)\alpha + \frac{n}{r}\beta + \frac{n}{r}\gamma$

Efficiency

Let $r = c = \sqrt{p}$, then

$$E_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}}$$

Strong scaling

Same story as before for $p \to \infty$:

$$E_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}} \sim p^{-1}$$

No strong scaling

Weak scaling

Constant memory $M = n^2/p$:

$$E_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}}$$

Weak scaling

Constant memory $M = n^2/p$:

$$E_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M}\frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}}\frac{(2\beta+\gamma)}{\gamma}}$$

Weak scaling

Constant memory $M = n^2/p$:

$$E_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}} = \frac{1}{1 + \frac{\log_2(p)}{2M}\frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}}\frac{(2\beta+\gamma)}{\gamma}}$$

Weak scaling:
for $p \to \infty$ this is $\approx 1/\log_2 p$:
only slowly decreasing.

LU factorizations

- Needs a cyclic distribution
- This is very hard to program, so:
- Scalapack, 1990s product, not extendible, impossible interface
- Elemental: 2010s product, extendible, nice user interface (and it is way faster)

Sparse linear algebra

Boundary value problems

Consider in 1D

$$\begin{cases} -u''(x) = f(x, u, u') & x \in [a, b] \\ u(a) = u_a,\, u(b) = u_b \end{cases}$$

in 2D:

$$\begin{cases} -u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) & x \in \Omega = [0, 1]^2 \\ u(\bar{x}) = u_0 & \bar{x} \in \delta\Omega \end{cases}$$

Approximation of 2nd order derivatives

Taylor series (write h for δx):

$$u(x+h) = u(x) + u'(x)h + u''(x)\frac{h^2}{2!} + u'''(x)\frac{h^3}{3!} + u^{(4)}(x)\frac{h^4}{4!} + u^{(5)}(x)\frac{h^5}{5!} + \cdots$$

and

$$u(x-h) = u(x) - u'(x)h + u''(x)\frac{h^2}{2!} - u'''(x)\frac{h^3}{3!} + u^{(4)}(x)\frac{h^4}{4!} - u^{(5)}(x)\frac{h^5}{5!} + \cdots$$

Subtract:

$$u(x+h) + u(x-h) = 2u(x) + u''(x)h^2 + u^{(4)}(x)\frac{h^4}{12} + \cdots$$

so

$$u''(x) = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} - u^{(4)}(x)\frac{h^4}{12} + \cdots$$

Numerical scheme:

$$-\frac{u(x+h) - 2u(x) + u(x-h)}{h^2} = f(x, u(x), u'(x))$$

(2nd order PDEs are very common!)

This leads to linear algebra

$$-u_{xx} = f \rightarrow \frac{2u(x) - u(x+h) - u(x-h)}{h^2} = f(x, u(x), u'(x))$$

Equally spaced points on $[0,1]$: $x_k = kh$ where $h = 1/(n+1)$, then

$$-u_{k+1} + 2u_k - u_{k-1} = -1/h^2 \, f(x_k, u_k, u'_k) \quad \text{for } k = 1, \ldots, n$$

Written as matrix equation:

$$\begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} f_1 + u_0 \\ f_2 \\ \vdots \end{pmatrix}$$

Matrix properties

- Very sparse, banded
- Symmetric (only because 2nd order problem)
- Sign pattern: positive diagonal, nonpositive off-diagonal (true for many second order methods)
- Positive definite (just like the continuous problem)
- Constant diagonals (from constant coefficients in the DE)

Sparse matrix in 2D case

Sparse matrices so far were tridiagonal: only in 1D case.
Two-dimensional: $-u_{xx} - u_{yy} = f$ on unit square $[0,1]^2$
Difference equation:

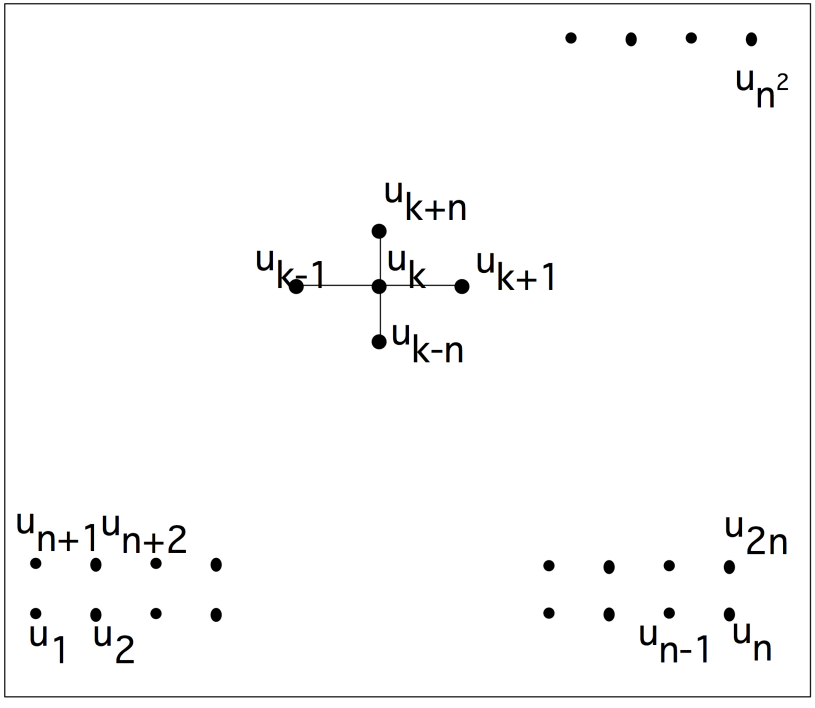
$$4u(x,y) - u(x+h,y) - u(x-h,y) - u(x,y+h) - u(x,y-h) = h^2 f(x,y)$$

$$4u_k - u_{k-1} - u_{k+1} - u_{k-n} - u_{k+n} = f_k$$

Consider a graph where $\{u_k\}_k$ are the edges
and (u_i, u_j) is an edge iff $a_{ij} \neq 0$.

The graph view of things

Poisson eq:



This is a graph!

This is the (adjacency) graph of a sparse matrix.

Sparse matrix from 2D equation

$$
\left(\begin{array}{ccccc|ccccc|c}
4 & -1 & & & \emptyset & -1 & & & & \emptyset & \\
-1 & 4 & 1 & & & & -1 & & & & \\
 & \ddots & \ddots & \ddots & & & & \ddots & & & \\
 & & \ddots & \ddots & -1 & & & & \ddots & & \\
\emptyset & & & -1 & 4 & \emptyset & & & & -1 & \\
\hline
-1 & & & & \emptyset & 4 & -1 & & & & -1 \\
 & -1 & & & & -1 & 4 & -1 & & & \\
 & \uparrow & \ddots & & & \uparrow & \uparrow & \uparrow & & & \uparrow \\
 & k-n & & & & k-1 & k & k+1 & & -1 & k+n \\
 & & & & -1 & & & & -1 & 4 & \\
\hline
 & & & & & \ddots & & & & & \ddots
\end{array}\right)
$$

Matrix properties

- Very sparse, banded
- Factorization takes less than n^2 space, n^3 work
- Symmetric (only because 2nd order problem)
- Sign pattern: positive diagonal, nonpositive off-diagonal (true for many second order methods)
- Positive definite (just like the continuous problem)
- Constant diagonals: only because of the constant coefficient differential equation
- Factorization: lower complexity than dense, recursion length less than N.

Realistic meshes