ETH zürich

spcl.inf.ethz.ch
@spcl_eth

D INFK

T. Hoefler, M. Pueschel

Lecture 9: Oblivious and non-oblivious algorithms

Teaching assistant: Salvatore Di Girolamo

Motivational video: https://www.youtube.com/watch?v=qx2dRIQXnbs

How many measurements are needed?

- **Measurements can be expensive!**
  - Yet necessary to reach certain confidence

- **How to determine the minimal number of measurements?**
  - Measure until the confidence interval has a certain acceptable width
  - For example, measure until the 95% CI is within 5% of the mean/median
  - Can be computed analytically assuming normal data
  - Compute iteratively for nonparametric statistics

- **Often heard: "we cannot afford more than a single measurement"**
  - E.g., Gordon Bell runs
  - Well, then one cannot say anything about the variance

    Even 3-4 measurement can provide very tight CI (assuming normality)

    Can also exploit repetitive nature of many applications

Experimental design

Rule 9: *Document all varying factors and their levels as well as the complete experimental setup (e.g., software, hardware, techniques) to facilitate reproducibility and provide interpretability.*

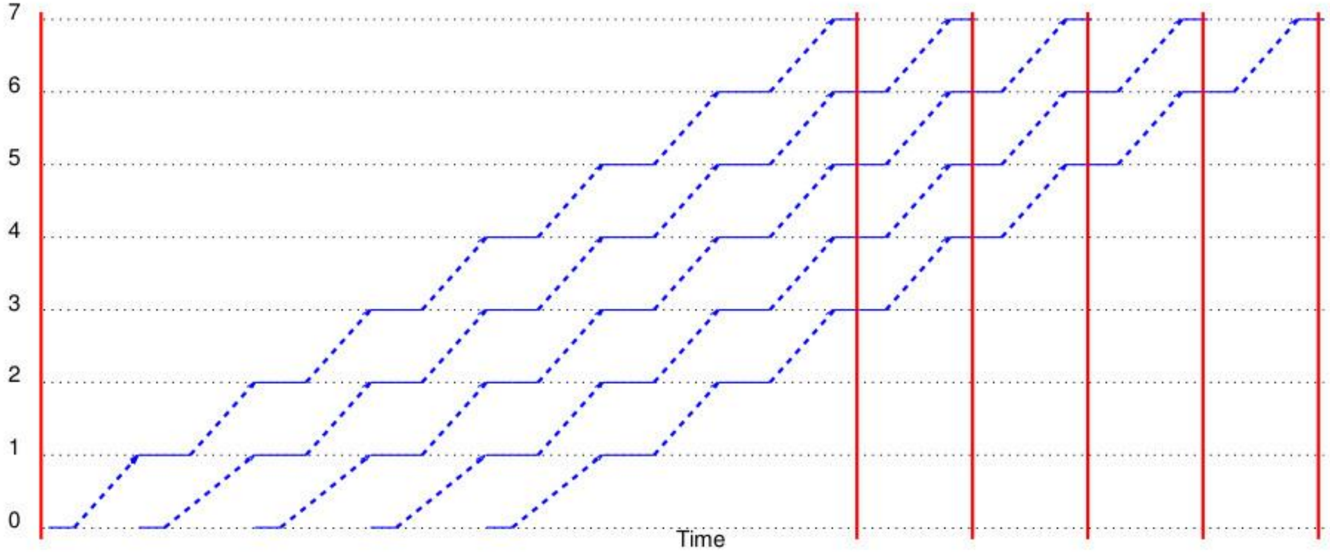
- **We recommend factorial design**
- **Consider parameters such as node allocation, process-to-node mapping, network or node contention**
  - If they cannot be controlled easily, use randomization and model them as random variable
- **This is hard in practice and not easy to capture in rules**

Time in parallel systems

My simple broadcast takes only one latency!

But I measured it so it must be true!

```
t = -MPI_Wtime();
for(i=0; i<1000; i++) {
  MPI_Bcast(…);
}
t += MPI_Wtime();
t /= 1000;
```



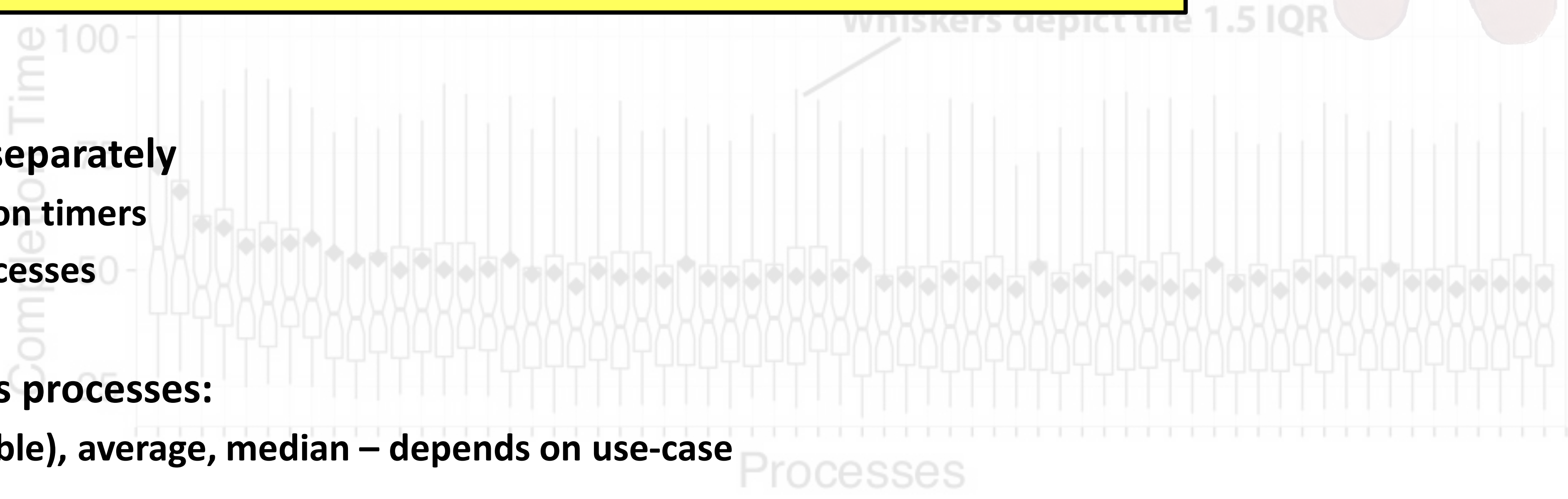
That’s nonsense!

…

Measure each operation separately!

Summarizing times in parallel systems!

Rule 10: *For parallel time measurements, report all measurement, (optional) synchronization, and summarization techniques.*



- **Measure events separately**
  - **Use high-precision timers**
  - **Synchronize processes**
- **Summarize across processes:**
  - **Min/max (unstable), average, median – depends on use-case**

Give times a meaning!

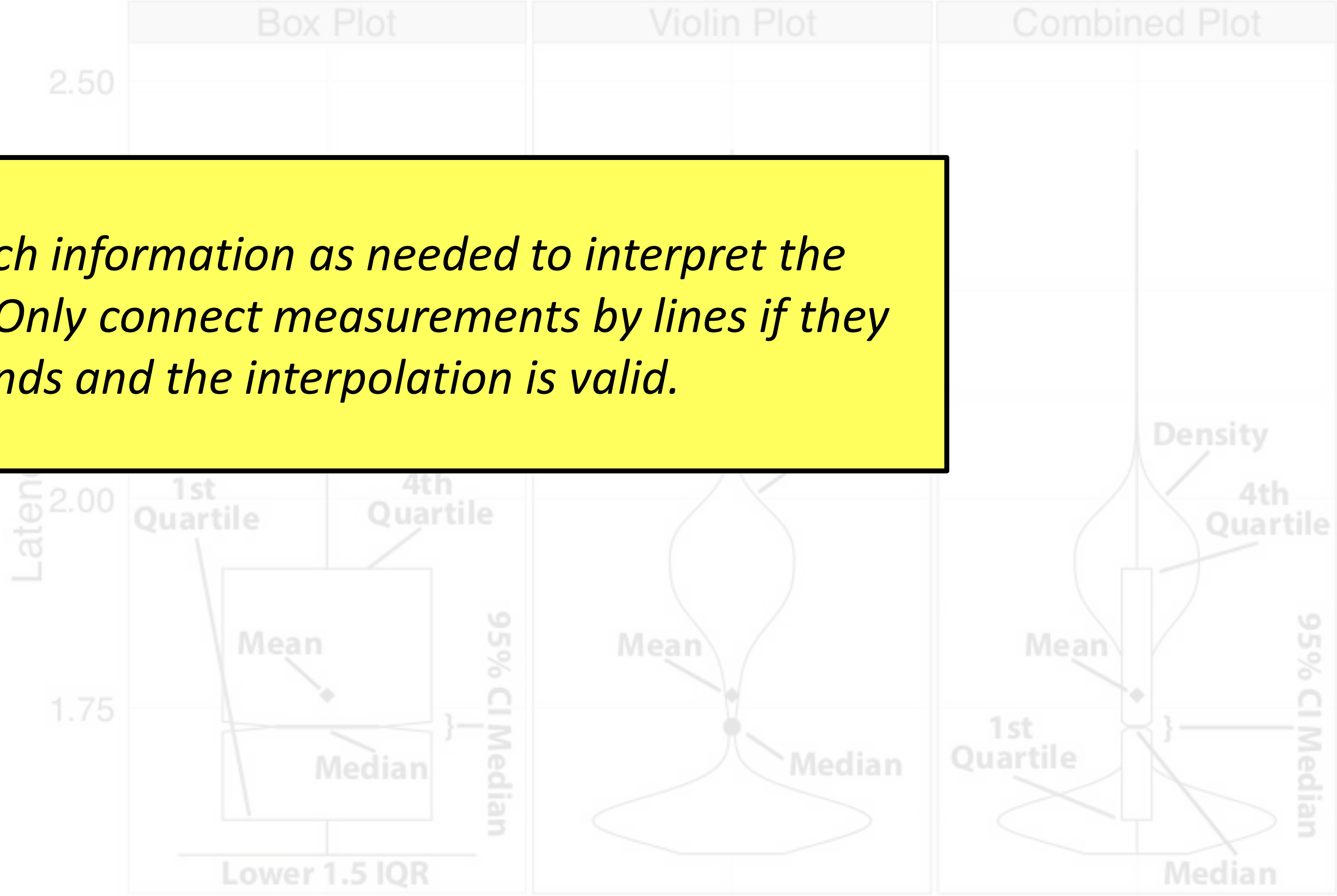
Rule 11: *If possible, show upper performance bounds to facilitate interpretability of the measured results.*

- **Model computer system as k-dimensional space**
  - Each dimension represents a capability
    Floating point, Integer, memory bandwidth, cache bandwidth, etc.
  - Features are typical rates
  - Determine maximum rate for each dimension
    E.g., from documentation or benchmarks
- **Can be used to proof optimality of implementation**
  - If the requirements of the bottleneck dimension are minimal

Plot as much information as possible!

My most common request was "show me the data"

This is how I should have presented the Dora results.



Rule 12: *Plot as much information as needed to interpret the experimental results. Only connect measurements by lines if they indicate trends and the interpolation is valid.*

Administrivia

- **Final project presentation: next Monday 12/17 during lecture**
  - Report will be due in January!
    Starting to write early is very helpful --- write – rewrite – rewrite (no joke!)

  - Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)
    What happened since the intermediate report?
    Focus on the key aspects (time is tight)!
    Try to wrap up – only minor things left for final report.
    Engage the audience ☺

  - Send slides by Sunday night (11:59pm Zurich time) to Salvatore!
    We will use a single (windows) laptop to avoid delays when switching
    Expect only Windows (powerpoint) or a PDF viewer
    The order of talks will again be randomized for fairness

Review of last lecture(s)

- **Impossibility of wait-free consensus with atomic registers**
  - "perhaps one of the most striking impossibility results in Computer Science" (Herlihy, Shavit)
- **Large-scale locks**
  - Scaling MCS to thousands of nodes with (MPI) RMA
- **Oblivious algorithms**
  - Execution oblivious vs. structural oblivious
  - Why do we care about obliviousness?
  - Strict optimality of work and depth – reduction ☺ – scan ☹

    Linear scan, tree scan, dissemination scan, surprising work-depth tradeoff W+D ≥ 2n-2
- **I/O complexity**
  - The red-blue pebble game (four rules: input, output, compute, delete)
  - S partitioning proof
  - Geometric arguments for dense linear algebra – example matrix multiplication

    Loomis Whitney inequality: $|V| \leq \sqrt{|V_x| + |V_y| + |V_z|}$ *(a set is smaller than sqrt of the sum of orthogonal projections)*
  - Simple recomputation – trade off I/O for compute

Learning goals for today

- **Strict optimality**
  - Work/depth tradeoffs and bounds
  - Applications of prefix sums

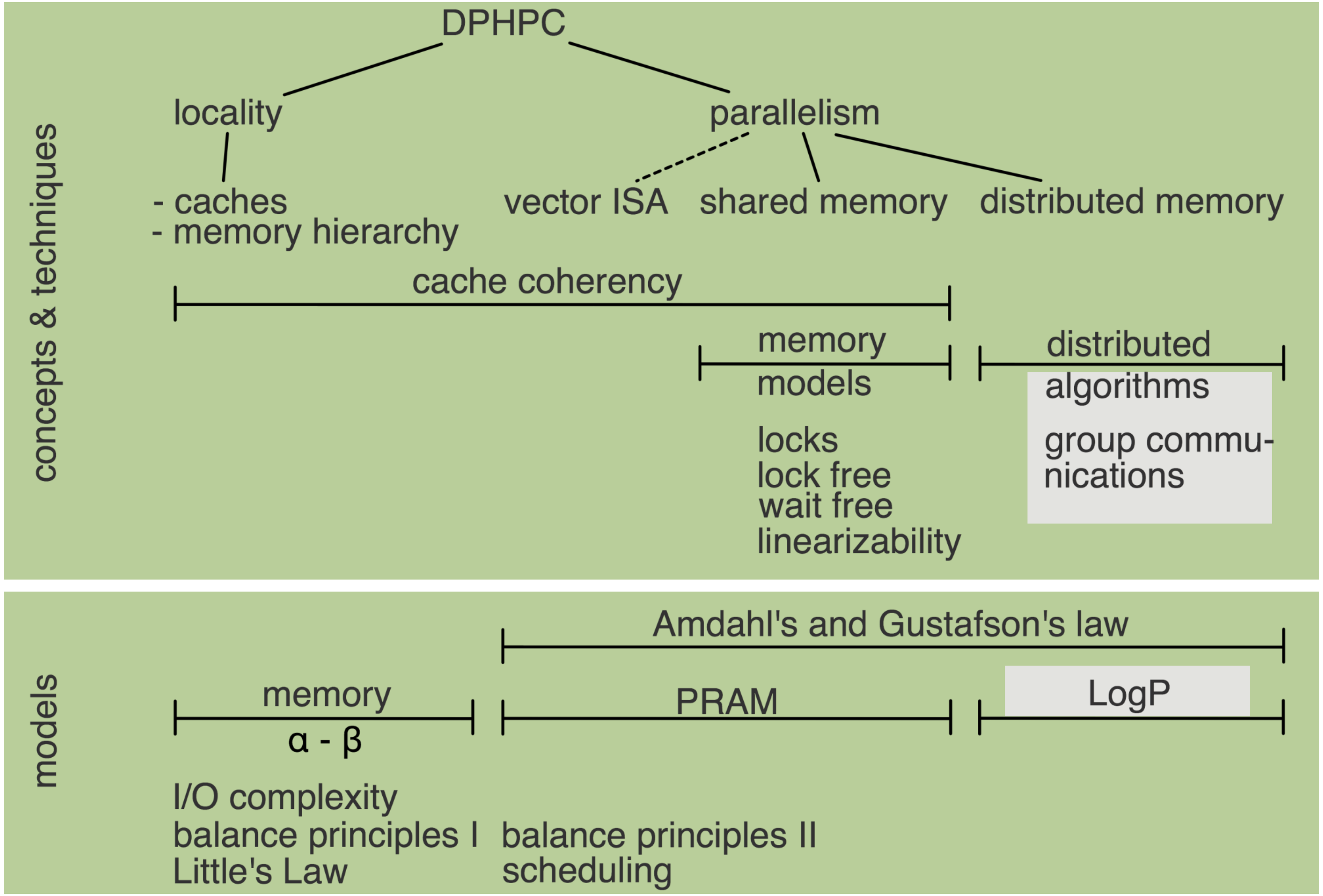
    Parallelize seemingly sequential algorithms

- **Oblivious graph algorithms**
  - Shortest paths
  - Connected components

- **Nonoblivious algorithms**
  - Sums and prefix sums on linked lists
  - Connected components

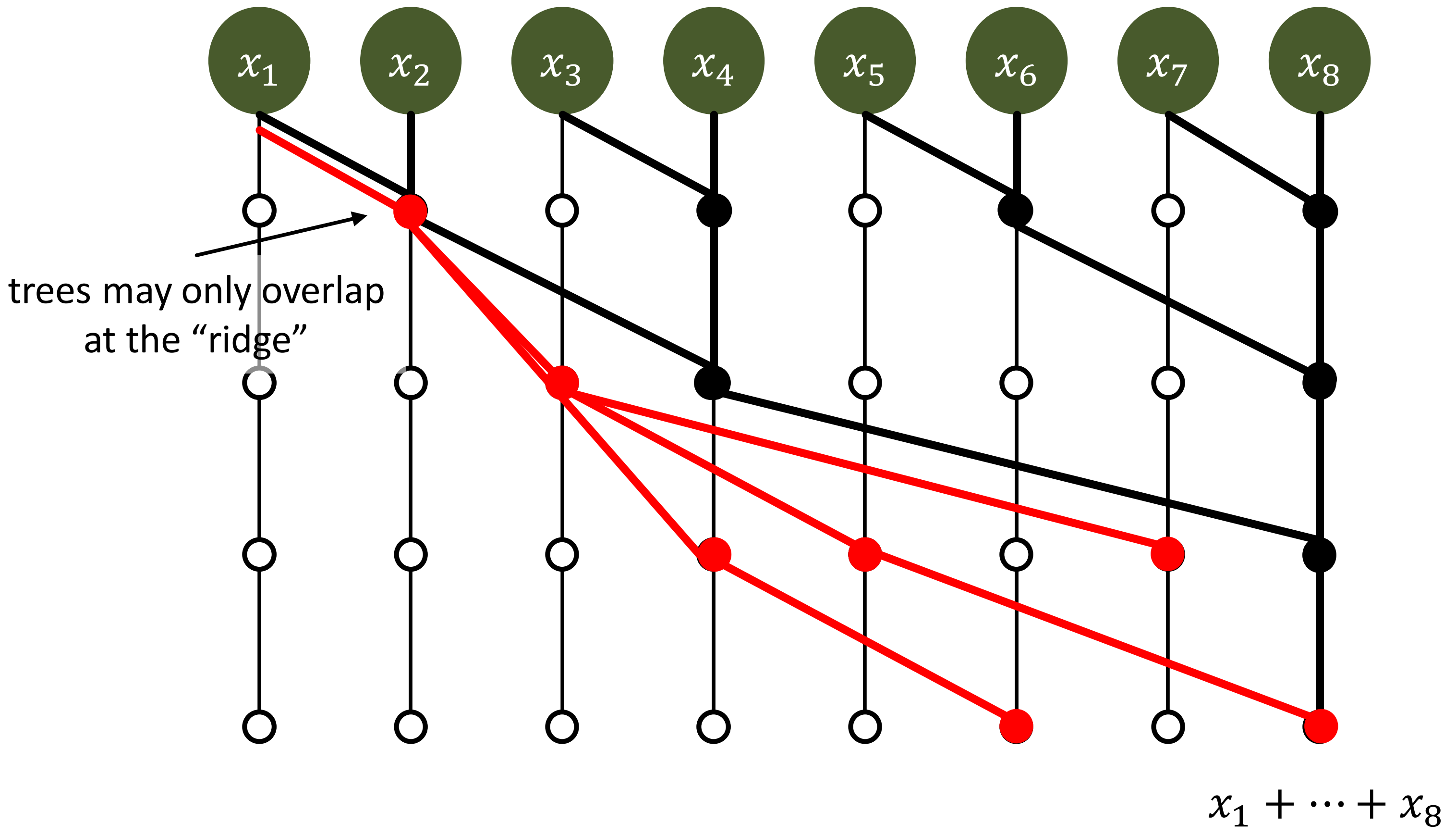
- **Distributed algorithms**
  - Broadcast in alpha-beta and LogP

DPHPC Overview



Recap: Work-depth tradeoff in parallel prefix sums

- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$



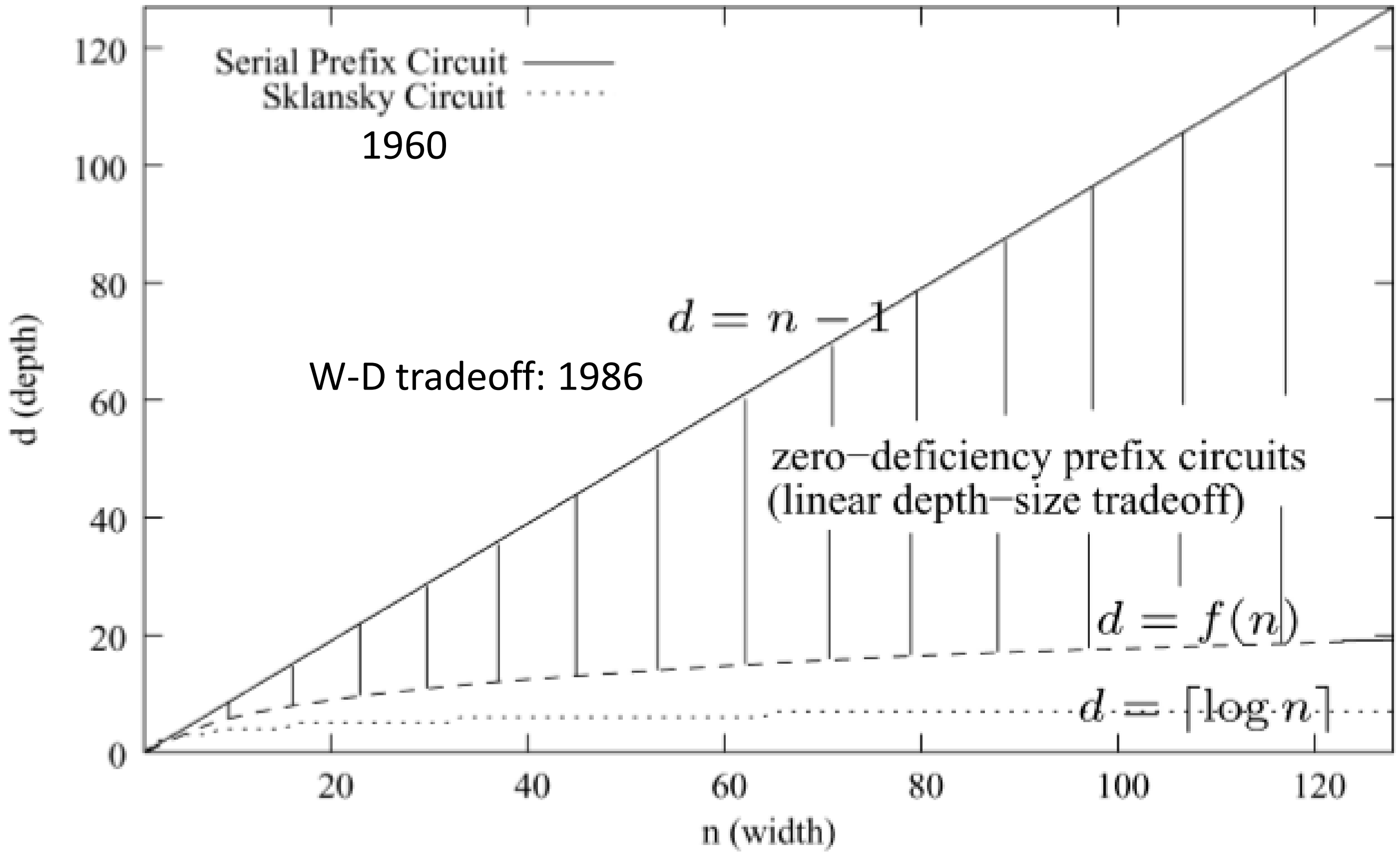
Output tree:
- leaves are all inputs, rooted at x_n
- binary due to binary operation
- $W = n - 1, D = D_o$

Input tree:
- rooted at x_1, leaves are all outputs
- not binary (simultaneous read)
- $W = n - 1$

Ridge can be at most D_o long!
Now add trees and subtract shared vertices:
$(n-1) + (n-1) - D_o = 2n - 2 - D_o \leq W$
q.e.d.

Work-Depth Tradeoffs and deficiency

“The deficiency of a prefix circuit c is defined as $\mathrm{def}(c) = W_c + D_c - (2n - 2)$”



Latest 2006 result for zero-deficiency construction for $n > F(D+3) - 1$ ($f(n)$ is inverse)

From Zhu et al.: “Construction of Zero-Deficiency Parallel Prefix Circuits”, 2006

Work- and depth-optimal constructions

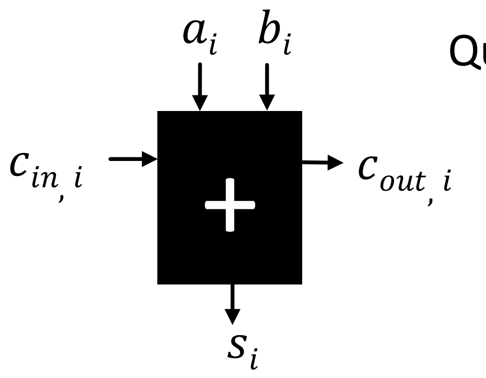
- **Work-optimal?**
  - Only sequential! Why?
  - $W = n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth
    $D = \lceil \log_2 n \rceil$, $W \leq 4n$
    Simple set of recursive construction rules (too boring for class, check 1980's paper if needed)
    Has an unbounded fan-out! May thus not be practical ☹

- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits

But why do we care about this prefix sum so much?

- **It's the simplest problem to demonstrate and prove W-D tradeoffs**
  - And it's one of the most important parallel primitives
- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!
- **Simple first example: binary adder – $s = a + b$ (n-bit numbers)**
  - Starting with single-bit (full) adder for bit i



Question: what are the functions for s_i and $c_{out,i}$?

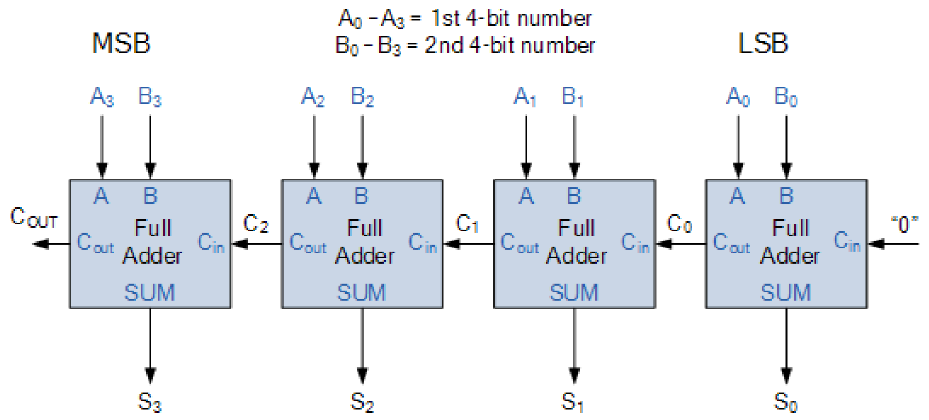
$$s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$$

$$c_{out,i} = a_i \text{ and } b_i \text{ or } c_{in,i} \text{ and } (a_i \text{ xor } b_i)$$

Show example 4-bit addition!

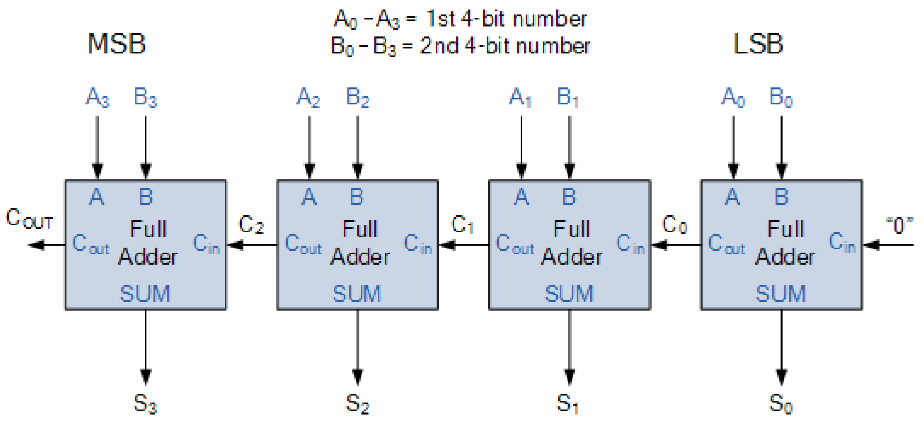
Question: what is work and depth?

Example 4-bit ripple carry adder



source: electronics-tutorials.ws

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

- **We only want s!**
  $$c_{out,i} = a_i \text{ and } b_i \text{ or } c_{in,i} \text{ and } (a_i \text{ xor } b_i)$$
  $$s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$$
  - Requires $c_{in,1}, c_{in,2}, \dots, c_{in,n}$ though ☹

- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0

    Encode state as 1-hot vector: $q_0 = \binom{1}{0}$, $q_1 = \binom{0}{1}$
  - Each full adder updates the carry bit state according to a_i and b_i

    State update is now represented by matrix operator, depending on a_i *and* b_i ($M_{a_i b_i}$):

    $$M_{00} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, M_{10} = M_{01} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$
  - Operator composition is defined on algebraic ring ({0, 1, or, and}) – i.e., replace "+" with "and" and "*" with "or"

    Prefix sum on the states computes now all carry bits in parallel!

Exercise: simplify!

- **Example: a=011, b=101 → M_{11}, M_{10}, M_{01}**
  - Scan computes: $M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$; $M_{11}M_{10} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$; $M_{11}M_{10}M_{01} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$ in parallel
  - All carry states and s_i can now be computed in parallel by multiplying scan result with q_0

Prefix sums as magic bullet for other seemingly sequential algorithms

- **Any time a sequential chain can be modeled as function composition!**
  - Let $f_1, \dots, f_n$ be an ordered set of functions and $f_0(x) = x$
  - Define ordered function compositions: $f_1(x); f_2(f_1(x)); \dots; f_n(\dots f_1(x))$
  - If we can write function composition $g(x) = f_i(f_{i-1}(x))$ as $g = f_i \circ f_{i-1}$ then we can compute $\circ$ with a prefix sum!

    We saw an example with the adder (M_{ab} were our functions)

- **Example: linear recurrence $\boldsymbol{f_i(x) = a_i f_{i-1}(x) + b_i}$ with $\boldsymbol{f_0(x)}$=x**
  - Write as matrix form $f_i\binom{x}{1} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} f_{i-1}\binom{x}{1}$
  - Function composition is now simple matrix multiplication!

    For example: $f_2\binom{x}{1} = \begin{pmatrix} a_2 & b_2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 & b_1 \\ 0 & 1 \end{pmatrix} f_0\binom{x}{1} = \begin{pmatrix} a_1 a_2 & a_2 b_1 + b_2 \\ 0 & 1 \end{pmatrix} \binom{x}{1}$

- **Most powerful! Homework:**
  - Parallelize tridiagonal solve (e.g., Thomas' algorithm)
  - Parallelize string parsing

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

    Iteration 3: 001 010 011 101 110 111

- **Now on n processors**
  - Each processor owns single k-bit number, each iteration

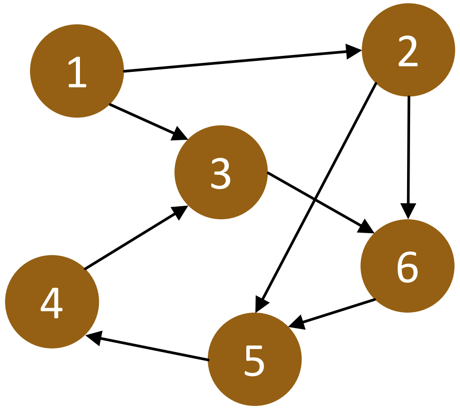
```
low = prefix_sum(!bit, sum)
high = n+1-backwards_prefix_sum(bit, sum)
new_idx = (bit == 0) : low ? high
b[new_idx] = a[i]
swap(a,b)
```

Show one example iteration!

Question: work and depth?

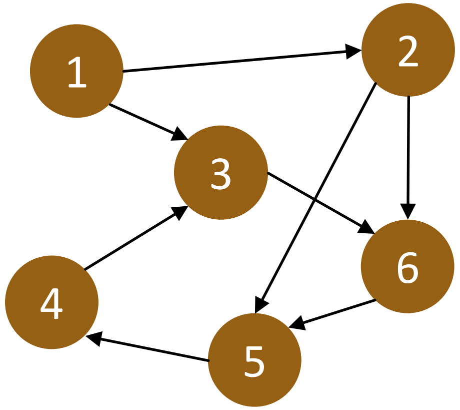
Oblivious graph algorithms

- **Seems paradoxical but isn't (may just not be most efficient)**
  - Use adjacency matrix representation of graph – "compute with all zeros"



0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

Unweighted graph – binary matrix



0	2	3	0	0	0
0	0	0	0	3	1
0	0	0	0	0	2
0	0	4	0	0	0
0	0	0	7	0	0
0	0	0	0	8	0

Weighted graph – general matrix

Algebraic semirings

- **A semiring is an algebraic structure that**
  - Has two binary operations called “addition” and “multiplication”
  - Addition must be associative ((a+b)+c = a+(b+c)) and commutative ((a+b=b+a)) and have an identity element
  - Multiplication must be associative and have an identity element
  - Multiplication distributes over addition (a*(b+c) = a*b+a*c) and multiplication by additive identity annihilates
  - Semirings are denoted by tuples (S, +, *, 0, 1)

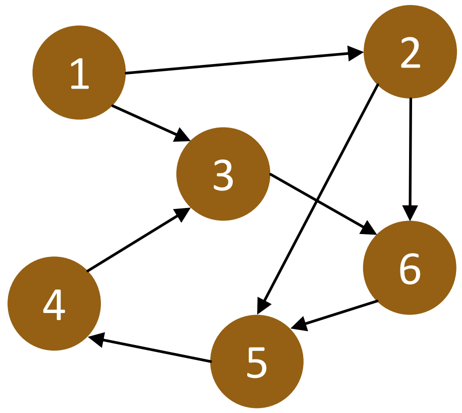
    *“Standard” ring of rational numbers: ($\mathbb{R}$, +, *, 0, 1)*

    Boolean semiring: ({0,1}, $\vee$, $\wedge$, 0, 1)

    Tropical semiring: ($\mathbb{R} \cup \{\infty\}$, min, +, ∞, 0) (also called min-plus semiring)

Oblivious shortest path search

- **Construct distance matrix from adjacency matrix by replacing all off-diagonal zeros with ∞**
- **Initialize distance vector d_0 of size n to ∞ everywhere but zero at start vertex**
  - E.g., $\mathbf{d_0} = (\infty, \mathbf{0}, \infty, \infty, \infty, \infty)^T$
    Show evolution when multiplied!



- **SSSP can be performed with n+1 matrix-vector multiplications!**
  - Question: total work and depth?
    $W = O(n^3), D = O(n \log n)$
  - Question: Is this good? Optimal?
    Dijkstra = $O(|E| + |V|\log|V|)$ ☹

0	∞	∞	∞	∞	∞
2	0	∞	∞	∞	∞
3	∞	0	4	∞	∞
∞	∞	∞	0	7	∞
∞	3	∞	∞	0	8
∞	1	2	∞	∞	0

- **Homework:**
  - Define a similar APSP algorithm with
    $W = O(n^3 \log n), D = O(\log^2 n)$

Oblivious connected components

- **Question: How could we compute the transitive closure of a graph?**
  - Multiply the matrix $(A + I)$ n times with itself in the Boolean semiring!
  - Why?

    Demonstrate that $(A + I)^2$ *has 1s for each path of at most length 1*

    By induction show that $(A + I)^k$ *has 1s for each path of at most length k*

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise
  - For each column (vertex) perform min-reduction to determine its component label!
  - Overall work and depth?

    $W = O(n^3 \log n)$, $D = O(\log^2 n)$

i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Many if not all graph problems have oblivious or tensor variants!

- **Not clear whether they are most efficient**
  - Efforts such as GraphBLAS exploit existing BLAS implementations and techniques

- **Generalizations to other algorithms possible**
  - Can everything be modeled as tensor computations on the right ring?
  - E. Solomonik, TH: *"Sparse Tensor Algebra as a Parallel Programming Model"*
  - Much of machine learning/deep learning is oblivious

- **Many algorithms get non-oblivious though**
  - All sparse algorithms are data-dependent!
  - E.g., use sparse graphs for graph algorithms on semirings (if $|E| < |V|^2/\log|V|$)
    May recover some of the lost efficiency by computing zeros!

- **Now moving to non-oblivious ☺**

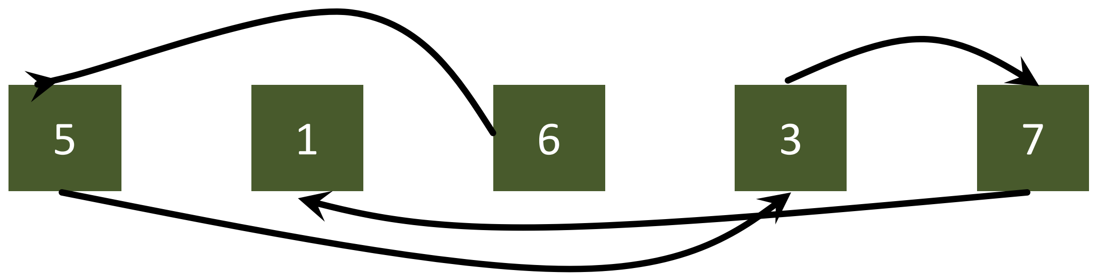
Nonoblivious parallel algorithms

- **Outline:**
  - Reduction on a linked list
  - Prefix sum on a linked list
  - Nonoblivious graph algorithms - connected components
  - Conflict graphs of bounded degree
- **Modeling assumptions:**
  - When talking about work and depth, we assume each loop iteration on a single PE is unit-cost (may contain multiple instructions!)

Reduction on a linked list

```
typedef struct elem {
  struct elem *next;
  int val
} elem;
```

- **Given: n values in linked list, looking for sum of all values**



- **Sequential algorithm:**

```
set S={all elems}
while (S != empty) {
  pick some i ∈ S;
  S = S – i.next;
  i.val += i.next.val;
  i.next = i.next.next;
}
```

A set $I \subset S$ is called an **independent set** if no two elements in I are connected!

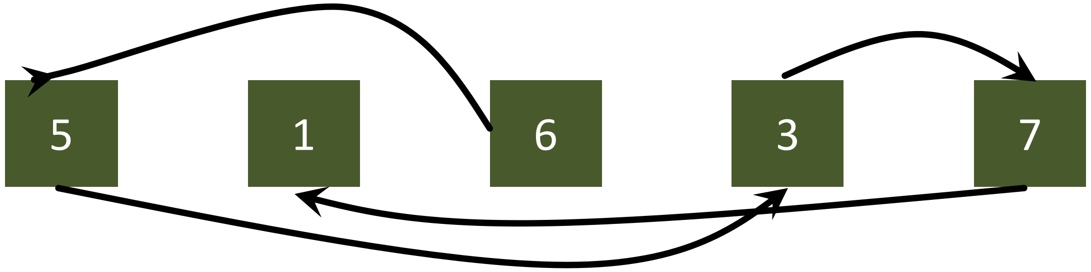
Are the following sets independent or not?

- {1}
- {1,5}
- {1,5,3}
- {7,6,5}
- {7,6,1}

Class question: What is the maximum size of an independent set of a linked list with n elements?

Parallel reduction on a linked list

- **Given: n values in linked list, looking for sum of all values**




```
typedef struct elem {
  struct elem *next;
  int val
} elem;
```

A subset $I \subset S$ is called an **independent set** if no two elements in I are connected!

- **Parallel algorithm:**

```
set S={all elems}
while (S != empty) {
  pick independent subset I ∈ S;
  for(each i ∈ I do in parallel) {
    S = S – i.next;
    i.val += i.next.val;
    i.next = i.next.next;
  }
}
```

Basically the same algorithm, just working on independent subsets!

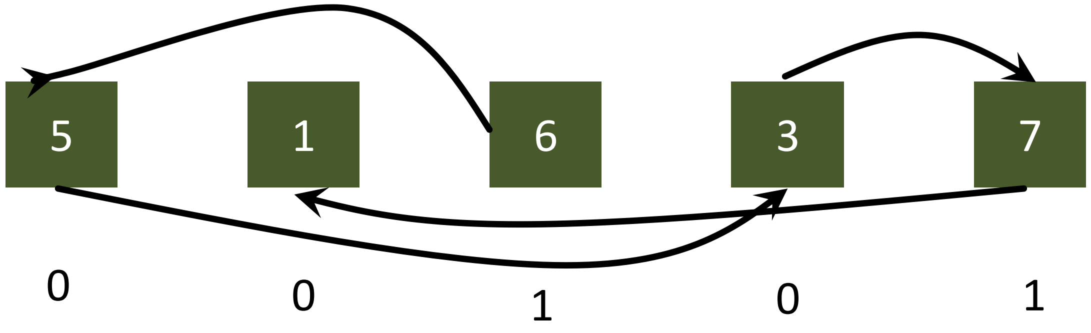
Class question: Assuming picking a maximum I is free, what are work and depth?

$$W = n - 1, D = \lceil \log_2 \mathrm{n} \rceil$$

Is this optimal?

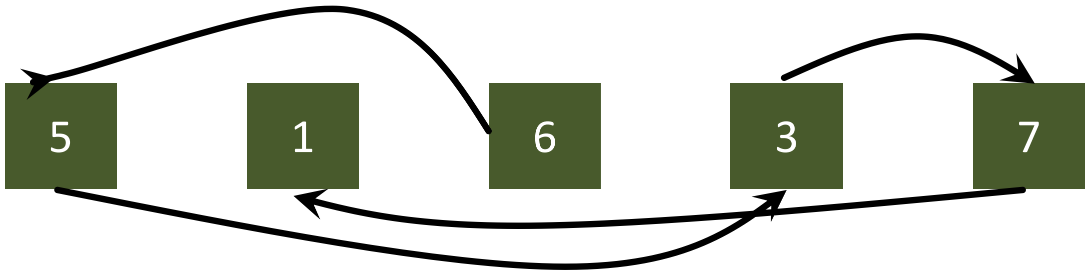
How to pick the independent set I?

- **That's now the whole trick!**
  - It's simple if all linked values are consecutive in an array – same as "standard" reduction!
    Can compute independent set up-front!
- **Irregular linked list though?**
  - Idea 1: find the order of elements → requires parallel prefix sum, D'oh!
  - Observation: if we pick $|I| > \lambda|V|$ in each iteration, we finish in logarithmic time!
- **Symmetry breaking:**
  - Assume p processes work on p consecutive nodes
  - How to find the independent set?
    They all look the same (well, only the first and last differ, they have no left/right neighbor)
    Local decisions cannot be made ☹
- **Introduce randomness to create local differences!**
  - Each node tosses a coin → 0 or 1
  - Let I be the set of nodes such that v drew 1 and $v.next$ drew 0!
    Show that I is indeed independent!
    What is the probability that $v \in I$? $P(v \in I) = \frac{1}{4}$



Optimizations

- **As the set shrinks, the random selection will get less efficient**
  - When p is close to n ($|S|$) then most processors will fail to make useful progress
  - Switch to a different algorithm
- **Recursive doubling!**




```
for (i=0; i ≤ ⌈log_2 n⌉; ++i) {
  for(each elem do in parallel) {
      elem.val += elem.next.val;
      elem.next = elem.next.next;
  }
}
```

Class question: What are work and depth?

$W = n\lceil\log_2 n\rceil, D = \lceil\log_2 \mathrm{n}\rceil$

  - Show execution on our example!
  - Algorithm computes prefix sum on the list!
    Result at original list head is overall sum

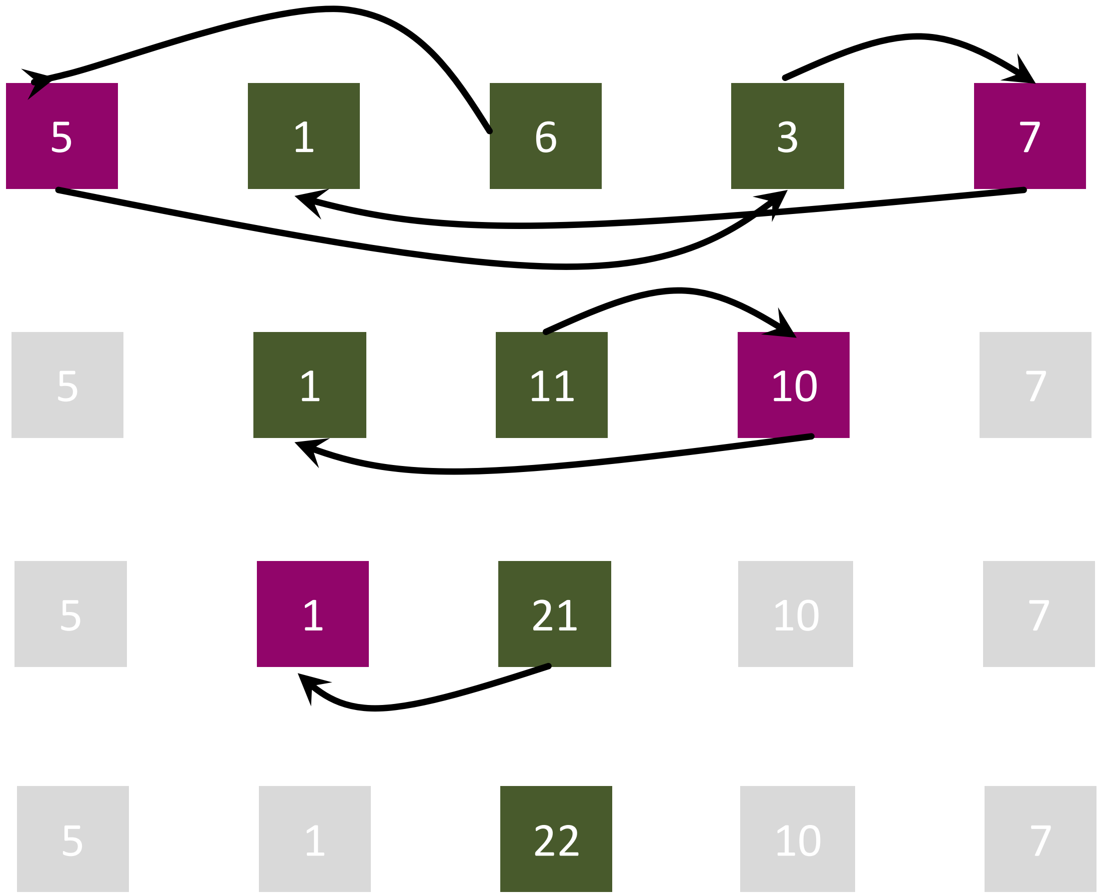
Prefix summation on a linked list

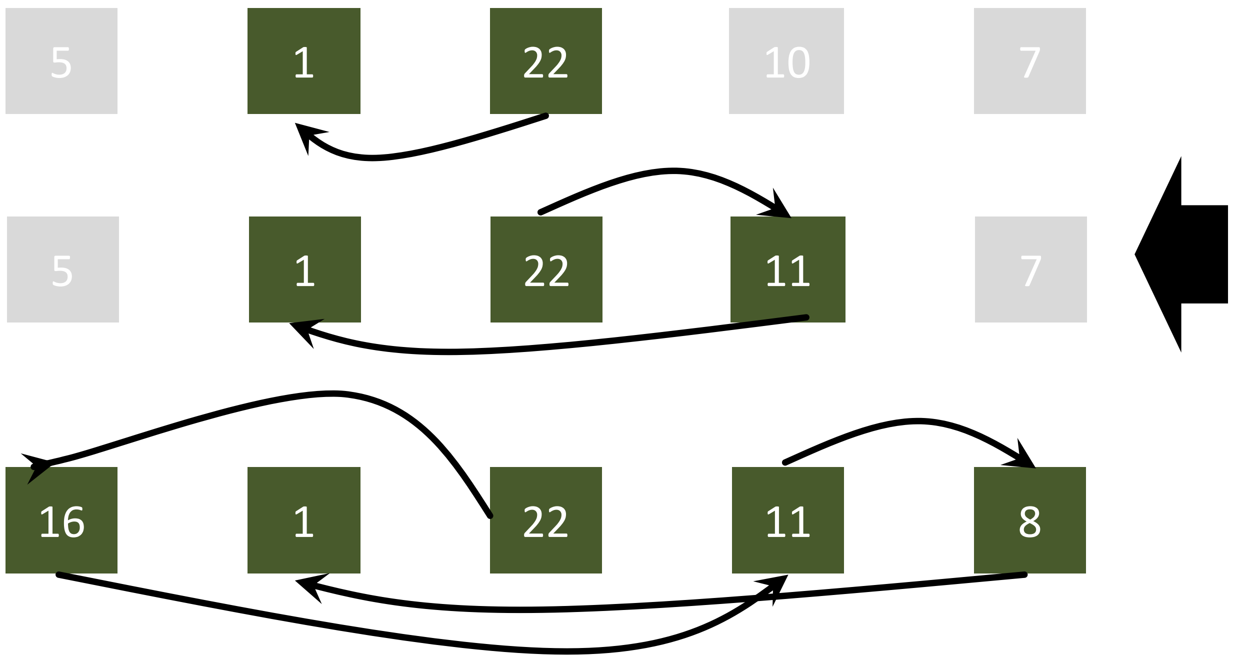
- **Didn't we just see it? Yes, but work-inefficient (if $p \ll n$)!**

  We extend the randomized symmetry-breaking reduction algorithms

  - First step: run the reduction algorithm as before
  - Second step: reinsert in reverse order of deletion

    When reinserting, add the value of their successor





Prefix summation on a linked list

- **Didn't we just see it? Yes, but work-inefficient (if $p \ll n$)!**

  We extend the randomized symmetry-breaking reduction algorithms
  - First step: run the reduction algorithm as before
  - Second step: reinsert in reverse order of deletion

    When reinserting, add the value of their successor

- **Class question: how to implement this in practice?**
  - Either recursion or a stack!
  - Design the algorithm as homework (using a parallel for loop)

Finding connected components as example

*A **connected component** of an undirected graph is a subgraph in which any two vertices are connected by a path and no vertex in the subgraph is connected to any vertices outside the subgraph. Each undirected graph G = (V,E) contains one or multiple (at most |V|) connected components.*

- **Straight forward and cheap to compute sequentially – question: how?**
  - Any traversal algorithm in work $O(|V| + |E|)$
    Seemingly trivial - becomes very interesting in parallel
  - Our oblivious semiring-based algorithm was $W = O(n^3 \log n), D = O(\log^2 n)$
    FAR from work optimality! Question: can we do better by dropping obliviousness?
- **Let's start simple – assuming concurrent read/write is free**
  - Arbitrary write wins
- **Concept of <u>supervertices</u>**
  - A supervertex represents a set of vertices in a graph

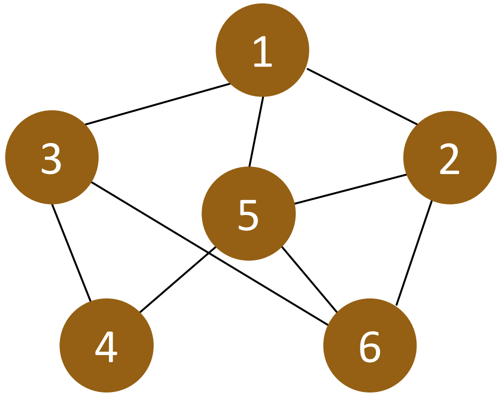
1. Initially, each vertex is a (singleton) supervertex
2. Successively merge neighboring supervertices
3. When no further merging is possible → each supervertex is a component

- Question is now only about the merging strategy

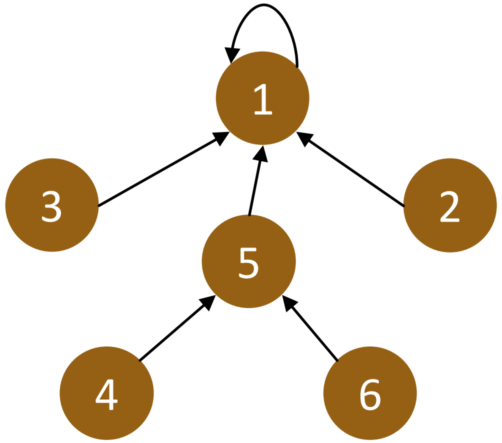
*A **fixpoint algorithm** proceeds iteratively and monotonically until it reaches a final state that is not left by iterating further.*

Shiloach/Vishkin's algorithm

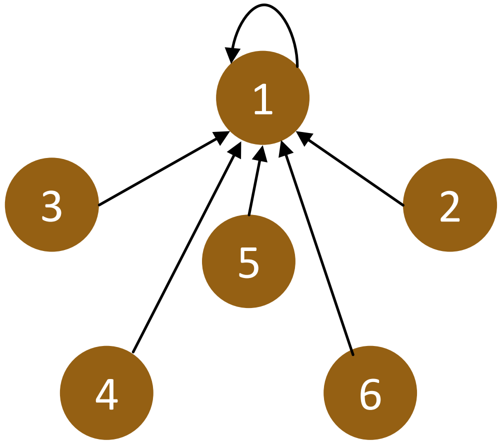
- **Pointer graph/forest:**
  - Define pointer array P, $P[i]$ is a pointer from i to some other vertex
  - We call the graph defined by P (excluding self loops) the pointer graph
  - During the algorithm, $P[i]$ forms a forest such that $\forall i$: $(i, P[i])$ there exists a path from i to $P[i]$ in the original graph!
  - Initially, all $P[i] = i$
  - The algorithm will run until each forest is a directed star pointing at the (smallest-id) root of the component
- **Supervertices:**
  - Initially, each vertex is its own supervertex
  - Supervertices induce a graph - S_i and S_j are connected iff $\exists(u, v) \in E$ with $u \in S_i$ and $v \in S_j$
  - A supervertex is represented by its tree in P



graph with single component



possible forest formed by P

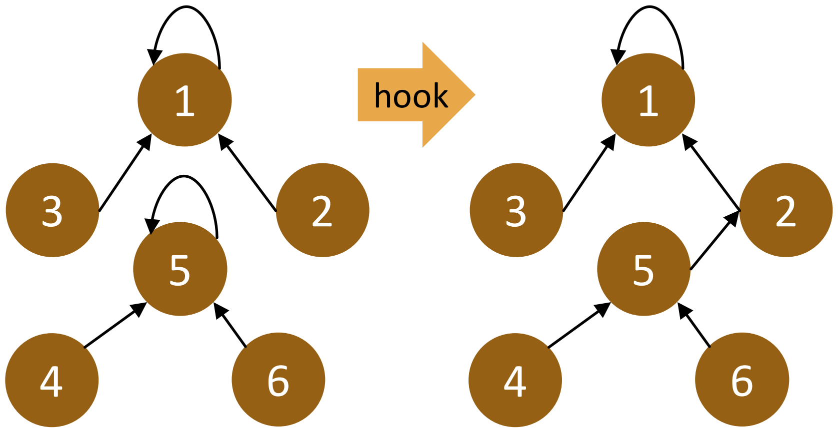


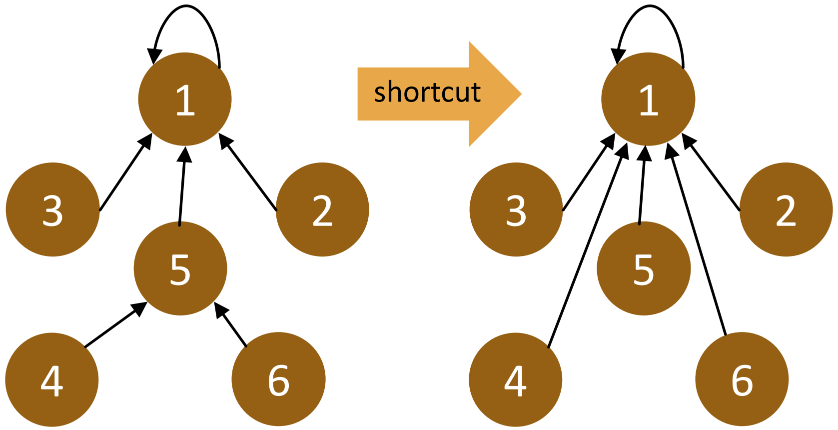
star formed by P

Shiloach/Vishkin's algorithm – key components

- **Algorithm proceeds in two operations:**
  - Hook – merge connected supervertices (must be careful to not introduce cycles!)
  - Shortcut – turn trees into stars

    Repeat two steps iteratively until fixpoint is reached!



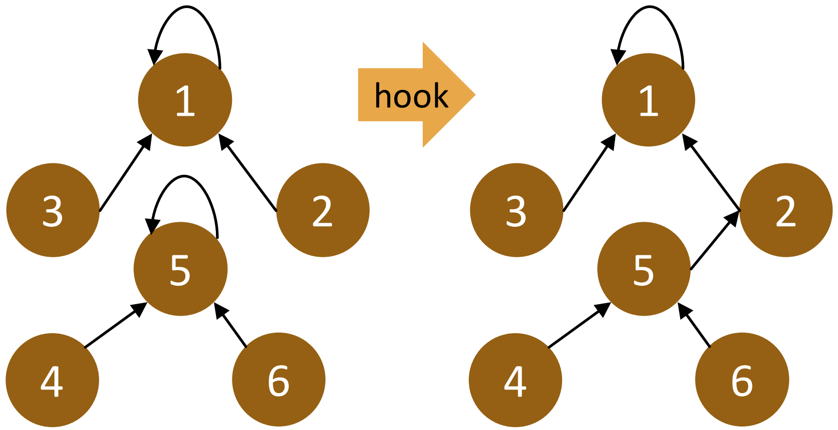


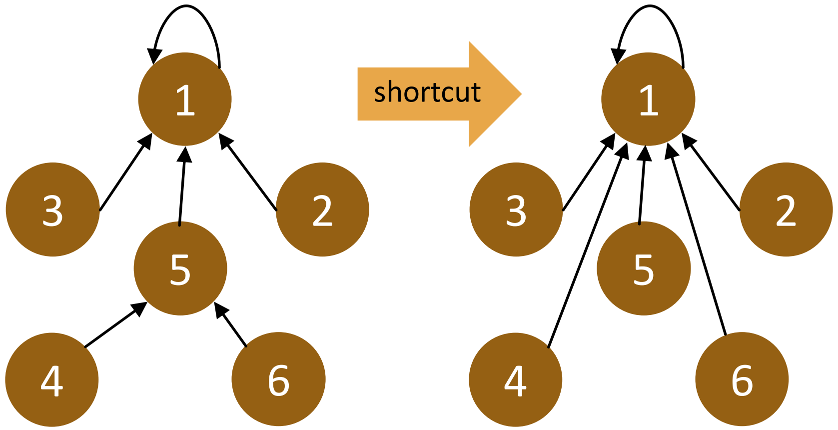
- **Correctness proofs:**
  - Lemma 1: The shortcut operation converts rooted trees to rooted stars. Proof: obvious
  - Theorem 1: The pointer graph always forms a forest (set of rooted trees). Proof: shortcut doesn't violate, hook works on rooted stars, connects only to smaller label star, no cycles

Shiloach/Vishkin's algorithm – key components

- **Algorithm proceeds in two operations:**
  - Hook – merge connected supervertices (must be careful to not introduce cycles!)
  - Shortcut – turn trees into stars

  Repeat two steps iteratively until fixpoint is reached!

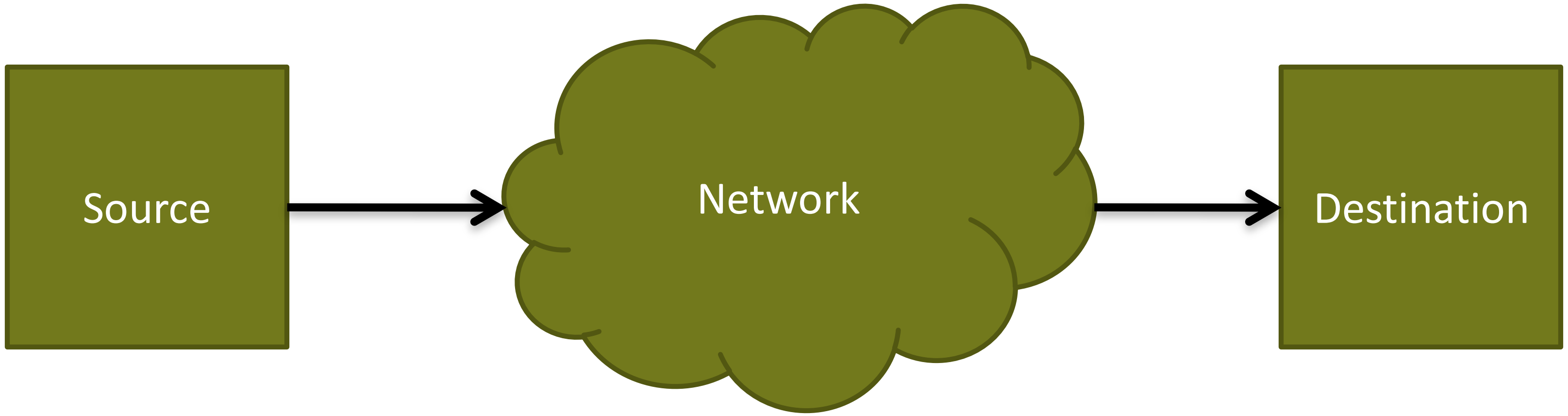




- **Performance proofs:**
  - Lemma 2: The number of iterations of the outer loop is at most $\log_2 n$. Proof: consider connected component, if it has two supervertices before hook, number of supervertices is halved, if no hooking happens, component is done
  - Lemma 2: The number of iterations of the inner loop in shortcut is at most $\log_2 n$. Proof: consider tree of height > 2 at some iteration, the height of the tree halves during that iteration
  - Corollary: Class question: work and depth? $W = O(n^2 \log n), D = O(\log^2 n)$ (assuming conflicts are free!)

Distributed networking basics

- **Familiar (non-HPC) network: Internet TCP/IP**
  - Common model:



- **Class Question: What parameters are needed to model the performance (including pipelining)?**
  - Latency, Bandwidth, Injection Rate, Host Overhead
  - What network models do you know and what do they model?

Remember: A Simple Model for Communication

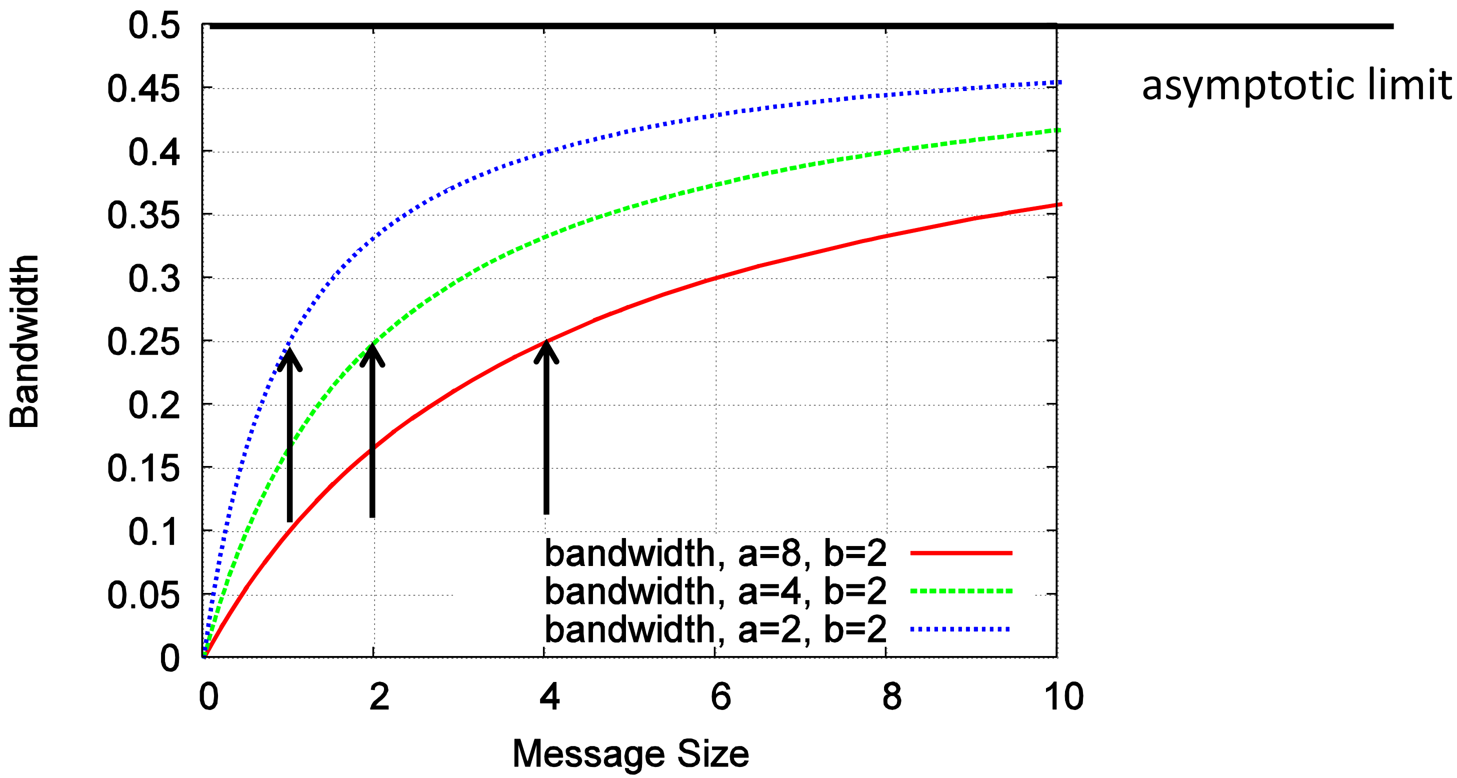
- **Transfer time** $T(s) = \alpha + \beta s$
  - α = startup time (latency)
  - β = cost per byte (bandwidth=1/β)

- **As s increases, bandwidth approaches** $1/\beta$ **asymptotically**
  - Convergence rate depends on α
  - $s_{\frac{1}{2}} = \alpha/\beta$

- **Assuming no pipelining (new messages can only be issued from a process after all arrived)**

Bandwidth vs. Latency

- $s_{\frac{1}{2}} = \alpha/\beta$ **is often used to distinguish bandwidth- and latency-bound messages**
  - $s_{\frac{1}{2}}$ is in the order of kilobytes on real systems



Quick Example

- **Simplest linear broadcast**
  - One process has a data item to be distributed to all processes

- **Linearly broadcasting s bytes among P processes:**
  - $T(s) = (P-1) \cdot (\alpha + \beta s) = O(P)$

- **Class question: Do you know a faster method to accomplish the same?**

k-ary Tree Broadcast

- **Origin process is the root of the tree, passes messages to k neighbors which pass them on**
  - k=2 -> binary tree

- **Class Question: What is the broadcast time in the simple latency/bandwidth model?**
  - $T(s) \approx \lceil \log_k P \rceil \cdot k(\alpha + \beta s)$ (for fixed k)

- **Class Question: What is the optimal k?**

  - $0 = \frac{k \ln P}{\ln k} \frac{d}{dk} = \frac{\ln\mathrm{P} \ln k - \ln P}{\ln^2 k} \rightarrow k = e = 2.71 \ldots$

  - Independent of P, α, βs? Really?

Faster Trees?

- **Class Question: Can we broadcast faster than in a ternary tree?**
  - Yes because each respective root is idle after sending three messages!
  - Those roots could keep sending!
  - Result is a k-nomial tree
    For k=2, it's a binomial tree

- **Class Question: What about the runtime?**
  - $T(s) = \lceil log_k(P) \rceil \cdot (k-1) \cdot (\alpha + \beta \cdot s) = \mathcal{O}(log(P))$

- **Class Question: What is the optimal k here?**
  - T(s) d/dk is monotonically increasing for k>1, thus k_{opt}=2

- **Class Question: Can we broadcast faster than in a k-nomial tree?**
  - $\mathcal{O}(log(P))$ is asymptotically optimal for s=1!
  - But what about large s?

Very Large Message Broadcast

- **Extreme case (P small, s large): simple pipeline**
  - Split message into segments of size z
  - Send segments from PE i to PE i+1

- **Class Question: What is the runtime?**
  - $T(s) = (P-2+s/z)(\alpha + \beta z)$

- **Compare 2-nomial tree with simple pipeline for α=10, β=1, P=4, s=10^6, and z=10^5**
  - 2,000,020 vs. 1,200,120

- **Class Question: Can we do better for given α, β, P, s?**
  - Derive by z $\quad z_{opt} = \sqrt{\frac{s\alpha}{(P-2)\beta}}$

- **What is the time for simple pipeline for α=10, β=1, P=4, s=10^6, z_{opt}?**
  - 1,008,964

Lower Bounds

- **Class Question: What is a simple lower bound on the broadcast time?**
  - $T_{BC} \geq \min\{\lceil \log_2(P) \rceil \alpha, s\beta\}$
- **How close are the binomial tree for small messages and the pipeline for large messages (approximately)?**
  - Bin. tree is a factor of $\log_2$(P) slower in bandwidth
  - Pipeline is a factor of P/$\log_2$(P) slower in latency
- **Class Question: What can we do for intermediate message sizes?**
  - Combine pipeline and tree → pipelined tree
- **Class Question: What is the runtime of the pipelined binary tree algorithm?**
  - $T \approx \left(\frac{s}{z} + \lceil \log_2 P \rceil - 2\right) \cdot 2 \cdot (\alpha + z\beta)$
- **Class Question: What is the optimal z?**
  - $z_{opt} = \sqrt{\frac{\alpha s}{\beta(\lceil \log_2 P \rceil - 2)}}$

Towards an Optimal Algorithm

- **What is the complexity of the pipelined tree with z_{opt} for small s, large P and for large s, constant P?**
  - Small messages, large P: s=1; z=1 (s≤z), will give O(log P)
  - Large messages, constant P: assume α, β, P constant, will give asymptotically O(sβ)

    Asymptotically optimal for large P and s but bandwidth is off by a factor of 2! Why?
- **Bandwidth-optimal algorithms exist, e.g., Sanders et al. *"Full Bandwidth Broadcast, Reduction and Scan with Only Two Trees".* 2007**
  - Intuition: in binomial tree, all leaves (P/2) only receive data and never send → wasted bandwidth
  - Send along two simultaneous binary trees where the leafs of one tree are inner nodes of the other
  - Construction needs to avoid endpoint congestion (makes it complex)

    Can be improved with linear programming and topology awareness

    (talk to me if you're interested)

Open Problems

- **Look for optimal parallel algorithms (even in simple models!)**
  - And then check the more realistic models
  - Useful optimization targets are MPI collective operations

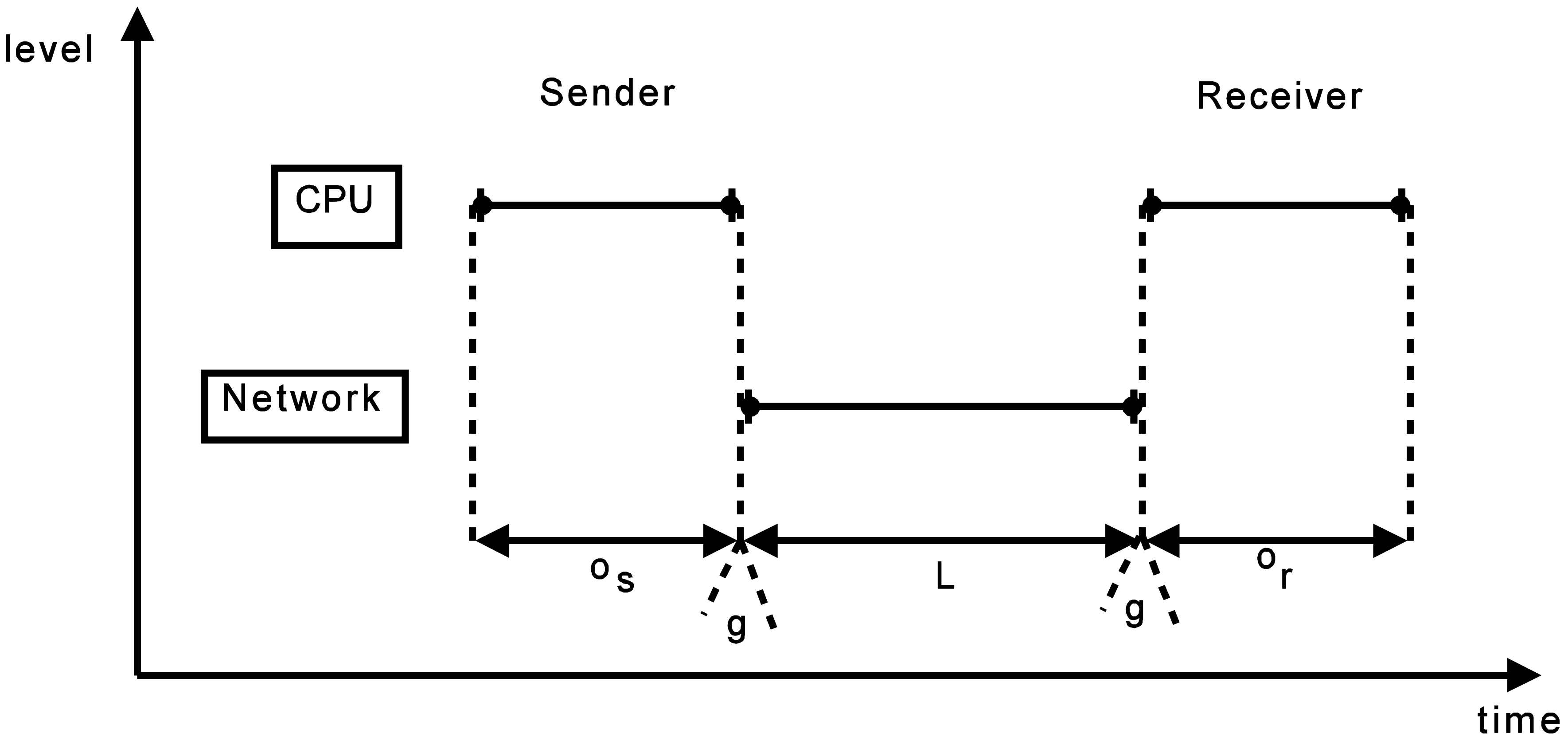
    Broadcast/Reduce, Scatter/Gather, Alltoall, Allreduce, Allgather, Scan/Exscan, …
  - Implementations of those (check current MPI libraries ☺)
  - Useful also in scientific computations

    Barnes Hut, linear algebra, FFT, …
- **Lots of work to do!**
  - Contact me for thesis ideas (or check SPCL) if you like this topic
  - Usually involve optimization (ILP/LP) and clever algorithms (algebra) combined with practical experiments on large-scale machines (10,000+ processors)

The LogP Model

- **Defined by four parameters:**
  - L: an upper bound on the latency, or delay, incurred in communicating a message containing a word (or small number of words) from its source module to its target module.
  - o: the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.
  - g: the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.
  - P: the number of processor/memory modules. We assume unit time for local operations and call it a cycle.

The LogP Model



Simple Examples

- **Sending a single message**
  - T = 2o+L
- **Ping-Pong Round-Trip**
  - T_{RTT} = 4o+2L
- **Transmitting n messages**
  - T(n) = L+(n-1)*max(g, o) + 2o

Simplifications

- **o is bigger than g on some machines**
  - g can be ignored (eliminates max() terms)
  - be careful with multicore!
- **Offloading networks might have very low o**
  - Can be ignored (not yet but hopefully soon)
- **L might be ignored for long message streams**
  - If they are pipelined
- **Account g also for the first message**
  - Eliminates “-1”

Benefits over Latency/Bandwidth Model

- **Models pipelining**
  - L/g messages can be "in flight"
  - Captures state of the art (cf. TCP windows)
- **Models computation/communication overlap**
  - Asynchronous algorithms
- **Models endpoint congestion/overload**
  - Benefits balanced algorithms

Example: Broadcasts

- **Class Question: What is the LogP running time for a linear broadcast of a single packet?**
  - $T_{lin} = L + (P-2) * \max(o,g) + 2o$
- **Class Question: Approximate the LogP runtime for a binary-tree broadcast of a single packet?**
  - $T_{bin} \leq \log_2 P * (L + \max(o,g) + 2o)$
- **Class Question: Approximate the LogP runtime for an k-ary-tree broadcast of a single packet?**
  - $T_{k\text{-}n} \leq \log_k P * (L + (k-1)\max(o,g) + 2o)$

Example: Broadcasts

- **Class Question: Approximate the LogP runtime for a binomial tree broadcast of a single packet (assume L > g!)?**
  - $T_{bin} \leq \log_2 P * (L + 2o)$
- **Class Question: Approximate the LogP runtime for a k-nomial tree broadcast of a single packet?**
  - $T_{k\text{-}n} \leq \log_k P * (L + (k-2)\max(o,g) + 2o)$
- **Class Question: What is the optimal k (assume o>g)?**
  - Derive by k: $0 = o * \ln(k_{opt}) - L/k_{opt} + o$ (solve numerically)

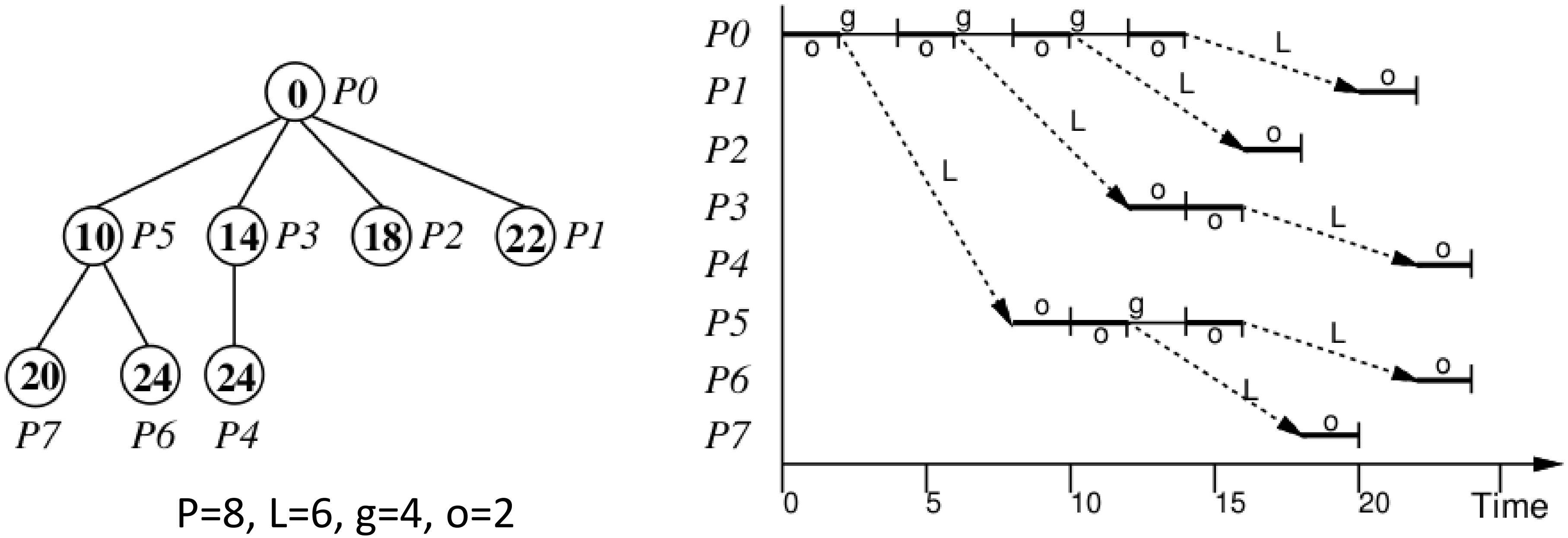
    For larger L, k grows and for larger o, k shrinks
  - Models pipelining capability better than simple model!

Example: Broadcasts

- **Class Question: Can we do better than k_{opt}-ary binomial broadcast?**
  - Problem: fixed k in all stages might not be optimal
  - We can construct a schedule for the optimal broadcast in practical settings
  - First proposed by Karp et al. in "Optimal Broadcast and Summation in the LogP Model"

Example: Optimal Broadcast

- **Broadcast to P-1 processes**
  - Each process who received the value sends it on; each process receives exactly once



P=8, L=6, g=4, o=2

Optimal Broadcast Runtime

- **This determines the maximum number of PEs (P(t)) that can be reached in time t**
- **P(t) can be computed with a generalized Fibonacci recurrence (assuming o>g):**

$$P(t) = \begin{cases} 1 : & t < 2o + L \\ P(t-o) + P(t-L-2o) : & \text{otherwise.} \end{cases} \qquad (1)$$

- **Which can be bounded by (see [1]):** $2^{\left\lfloor \frac{t}{L+2o} \right\rfloor} \leq P(t) \leq 2^{\left\lfloor \frac{t}{o} \right\rfloor}$
  - A closed solution is an interesting open problem!

[1]: Hoefler et al.: “Scalable Communication Protocols for Dynamic Sparse Data Exchange” (Lemma 1)

The Bigger Picture

- **We learned how to program shared memory systems**
  - Coherency & memory models & linearizability
  - Locks as examples for reasoning about correctness and performance
  - List-based sets as examples for lock-free and wait-free algorithms
  - Consensus number
- **We learned about general performance properties and parallelism**
  - Amdahl's and Gustafson's laws
  - Little's law, Work-span, …
  - Balance principles & scheduling
- **We learned how to perform model-based optimizations**
  - Distributed memory broadcast example with two models
- **What next? MPI? OpenMP? UPC?**
  - Next-generation machines "merge" shared and distributed memory concepts → Partitioned Global Address Space (PGAS)

If you're interested in any aspect of parallel algorithms, programming, systems, or large-scale computing and are looking for a thesis, let us know! (and check our webpage http://spcl.inf.ethz.ch/SeMa)