Parallel and Distributed Algorithms

Wintersemester 09/10

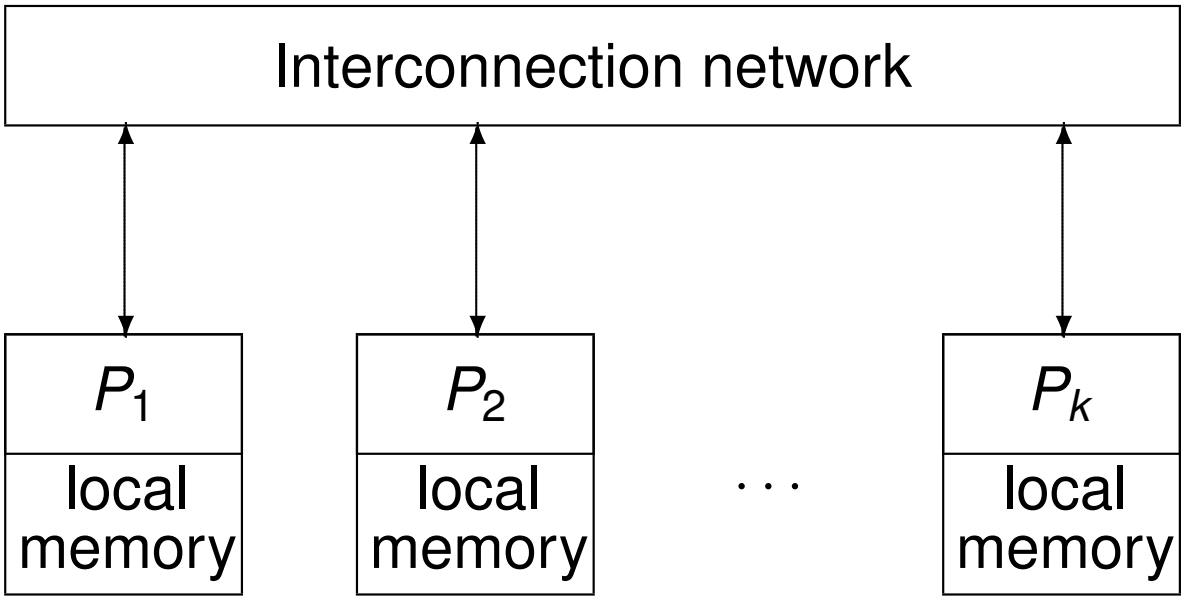
Welcome!

How to design parallel algorithms?

- General methods:
  - Communication patterns such as trees, meshes, hypercube architectures.
  - Design principles such as divide & conquer as well as dynamic programming.
- Algorithms for specific problems:
  - Computational linear algebra: matrix-vector and matrix-matrix product, solving linear systems, Fast Fourier Transform.
  - Monte Carlo methods.
  - Algorithms for hard problems: backtracking, branch & bound, alpha-beta pruning.
  - problems for lists and trees: prefix problems, list ranking and tree contraction.
  - problems for graphs: connected components, spanning trees, shortest paths, coloring.
- Load Balancing.

A Multicomputer

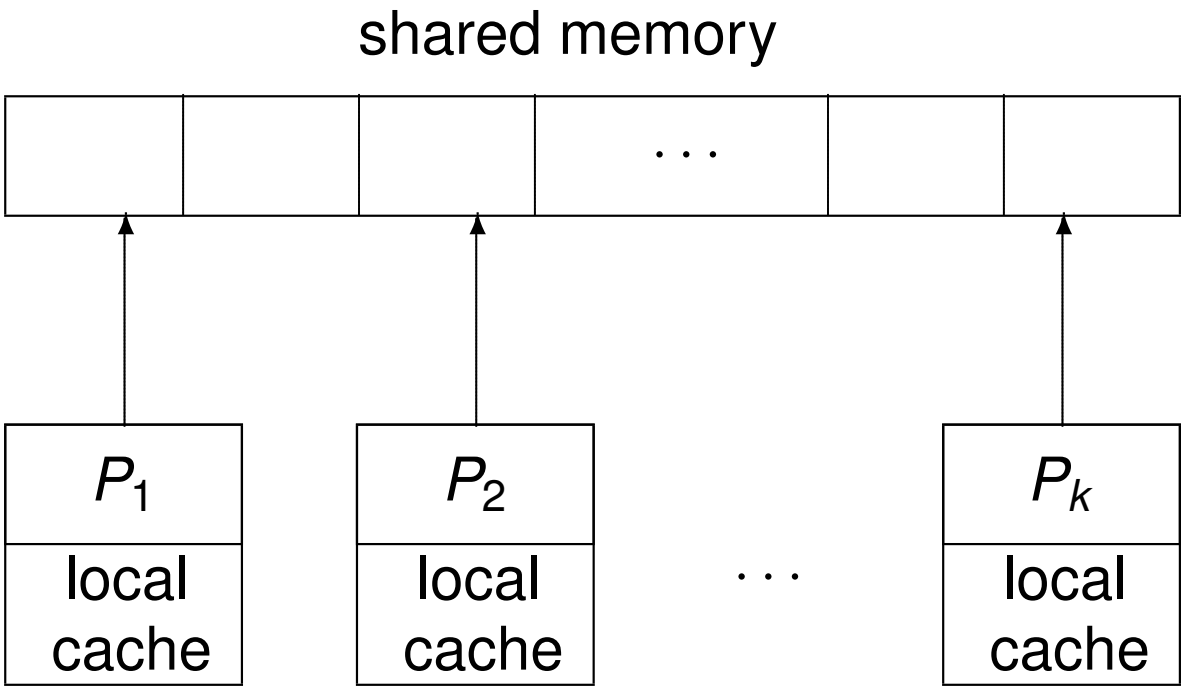
- consists of a cluster of processors (such as workstations or personal computers) with a distributed memory.
- Processors send point-to-point messages. Messages are routed by an interconnection network of switches.
- Network technologies: Gigabit Ethernet, Myrinet or InfiniBand.
- Cheap. Large aggregate memory capacity. Slow communication.



k processors in a distributed memory model.

A Multiprocessor

- consists of several CPUs which access their local cache as well as a shared memory.
- Relatively fast communication. However, due to hardware restrictions, only few processors are supported.



k processors in a shared memory model.

Hybrid Systems

- Multicomputer: the speed of communication links grows.
- Multiprocessors: multicore machines have become mass products.
- Hybrid systems: multicomputers composed of multiprocessors are the consequence. MPI 2 supports hybrid systems.

  CSC is a cluster composed of dual-core CPUs.

What is a good parallel algorithm?
Analyzing parallel algorithms

- Determine the speedup in comparison to good sequential algorithms.
- How does an algorithm scale with a growing number of processes? Does the computing time decrease accordingly?
- How large is the communication cost? Is it possible to cover up communication with local computation?

When is it possible to come up with really fast parallel algorithms?

We introduce the notion of $\mathcal{P}$-completeness.

- $\mathcal{P}$ is the class of all problems which have efficient sequential algorithms.
- If any $\mathcal{P}$-complete problem has a really fast parallel algorithm, then all problems in $\mathcal{P}$ have really fast parallel algorithms.
- Consequence: In all likelihood $\mathcal{P}$-complete problems have no really fast algorithms and we shouldn't be looking for fast parallel algorithms!
- Which problems are $\mathcal{P}$- complete? Linear programming, depth-first-search, circuit evaluation and many more.

Time Table

- Contents:
  - The first part concentrates on message passing and $\mathcal{P}$-completeness.
    Goals: Design and evaluate parallel algorithms for multicomputers.
  - The second part deals with shared memory programming.
    Goals: Study the effect of shared memory.
  - In the final part we explore the limits of parallelization.
- For CSC students: all classes during the first half of the semester are relevant. Tutorials do not have to be attended.
- A parallel programming course based on this course is offered next semester!

Required Background

- A Vordiplom or Bachelor degree in computer science is fully sufficient.
- Otherwise, the following subjects are helpful:
  - Introductory programming courses,
  - an introductory algorithms course as well as
  - courses on linear algebra, calculus and elementary probability theory.
- Chapter 1 of the lecture notes contains all mathematical prerequisites.

Literature

- Message Passing:
  - M.J. Quinn, Parallel Programming in C with MPI and OpenMP, *MCGraw Hill*, 2004.
  - A. Grama, A. Gupta, G. Karypis und V. Kumar, Introduction to parallel Computing, second edition, *Addison-Wesley*, 2003.
- Shared Memory:
  - J. Já Já , An Introduction to Parallel Algorithms, *Addison-Wesley*, 1992.
  - R.M. Karp und V. Ramachandran, Parallel algorithms for shared memory machines, in J van Leeuven (Ed.): Handbook of Theoretical Computer Science A, Kapitel 17, pp. 869-941, *Elsevier Science Publishers*, 1990.
- P-completeness: R. Greenlaw, H.J. Hoover und W.L. Ruzzo, Limits to Parallel Computation, *Oxford University Press*, 1995.
- Lecture Notes: See website of class.

Asymptotic Notation

- How does computation time grow with increased input size?
- How does computation time vary in dependence on the number of processes?
- For large input sizes the asymptotics takes over. Therefore, for functions $f, g : \mathbb{N} \to \mathbb{R}$ define
  - $f = O(g)$ iff $f(n) \leq c \cdot g(n)$ for all $n \geq N$, where N and c are constants.
  - $f = \Omega(g)$ iff $g = O(f)$.
  - $f = \Theta(g)$ iff $f = O(g)$ and $g = O(f)$.
  - $f = o(g)$ iff $\lim_{n\to\infty} f(n)/g(n) = 0$.
- Limits and asymptotic growth:
  - if $0 \leq \lim_{n\to\infty} f(n)/g(n) < \infty$, then $f = O(g)$.
  - if $0 < \lim_{n\to\infty} f(n)/g(n) < \infty$, then $f = \Theta(g)$.

Graphs

- An undirected graph is a pair $G = (V, E)$ consisting of a set V of nodes and a set E of edges.
- Any edge $e \in E$ is represented by a two-element subset $e = \{u, v\}$: u and v are the endpoints of e. We say that u and v are neighbors.
- In a directed graph edges are represented by ordered pairs (u, v) of nodes. We say that v is an immediate successor of u.

Trees and Forests

- A walk of length k in G is a sequence $v_0, e_1, v_1 \ldots, e_k, v_k$ of nodes and edges such that $e_i = \{v_{i-1}, v_i\}$.
- A walk without repeated nodes is a path. The distance between two nodes is the minimum length of a path connecting them.
- A cycle of length k is a walk $v_0, \ldots, v_k$ such that $v_0 = v_k$ and $v_0, \ldots, v_{k-1}$ is a path.
- An undirected graph Graph G is connected, if there is a path connecting any two of its nodes.
- A forest is a graph without cycles. A tree is a connected graph without cycles.

Communication Patterns

- How to distribute information efficiently? Who should talk to whom?
- We model a communication pattern by an undirected graph $G = (V, E)$ of processes where we introduce an edge between processes p and q, whenever p and q communicate.
- When is G a "good" communication pattern? Important parameters are its degree, diameter and bisection width.
  - The degree of G is the maximal number of neighbors of a node.
  - The diameter of G is the maximal distance between any two nodes. The diameter determines the time to route information between distant nodes. Thus keep the diameter small.

Bisection Width

The bisection width of G is the minimal number of edges, whose removal disconnects the graph into two halves with sizes differing by at most one.

- Bisection width b implies that b edges have to carry all information from one half to the other.
- If b is small, then few links have to carry all messages.
- The bisection width should be as large as possible. However a large bisection width comes with the price of complex communications.
- The bisection width of trees is small. It increases for meshes with increasing dimension.

Trees

- T_k is the ordered, complete binary tree of depth k.
  Any interior node has a left and a right child.
- The good news: the degree of T_k is two, its diameter is $2k$.
- The bad news: the bisection width is one, since we may remove one of the two edges of the root.

  if nodes outside of the subtree $T_k(v)$ with root v require information held by nodes of $T_k(v)$, then this information has to travel across bottleneck v.

What are Trees Good for?

n processors are given, where the ith processor knows key x_i. Determine the maximal key. .

We use the tournament approach.

- Assign the keys to the leaves of T_k such that each leaf obtains $\frac{n}{2^k}$ keys.
- All "leaves" determine their maximum in parallel.
- Then the respective maximal keys are passed upwards from the children to their parents, who determine their maximum and pass it upwards themselves.
- We set $p = 2^{k+1} - 1$ and we have computed the maximum of n keys with p processors in $O(\log_2 p + \frac{n}{p})$ compute steps and $\log_2 p$ communication steps. Is it possible to obtain the same performance with just $p = 2^k$ processors?

Prefix Problems

The tournament approach has many more applications.

A set U and an operation $* : U \times U \to U$ is given. We demand that operation $*$ is associative, i.e., we require that

$$x * (y * z) = (x * y) * z$$

holds for all $x, y, z \in U$. The prefix problem for the input sequence $x_1, x_2, \ldots, x_n \in U$ is to determine all prefixes

$$\begin{array}{l} x_1 \\ x_1 * x_2 \\ x_1 * x_2 * x_3 \\ \quad \vdots \\ x_1 * x_2 * \cdots * x_n \end{array}$$

Applications I

- An array A of m cells is given and we assume that a cell is either empty or stores a value. We would like to remove all empty cells. We set $U = \mathbb{N}$ and

$$x_i := \begin{cases} 1 & A[i] \text{ is non-empty,} \\ 0 & \text{otherwise.} \end{cases}$$

  If cell i is not empty, then $A[i]$ has to be moved to cell $x_1 + x_2 + \cdots + x_i$.

- The problem of determining the maximum is a prefix problem, if we define operation $*$ by

$$x * y = \max\{x, y\}.$$

  $x_1 * \cdots * x_n$ is the maximum of $x_1, \ldots, x_n$.

- The reals with addition or multiplication define a prefix problem.

Applications II

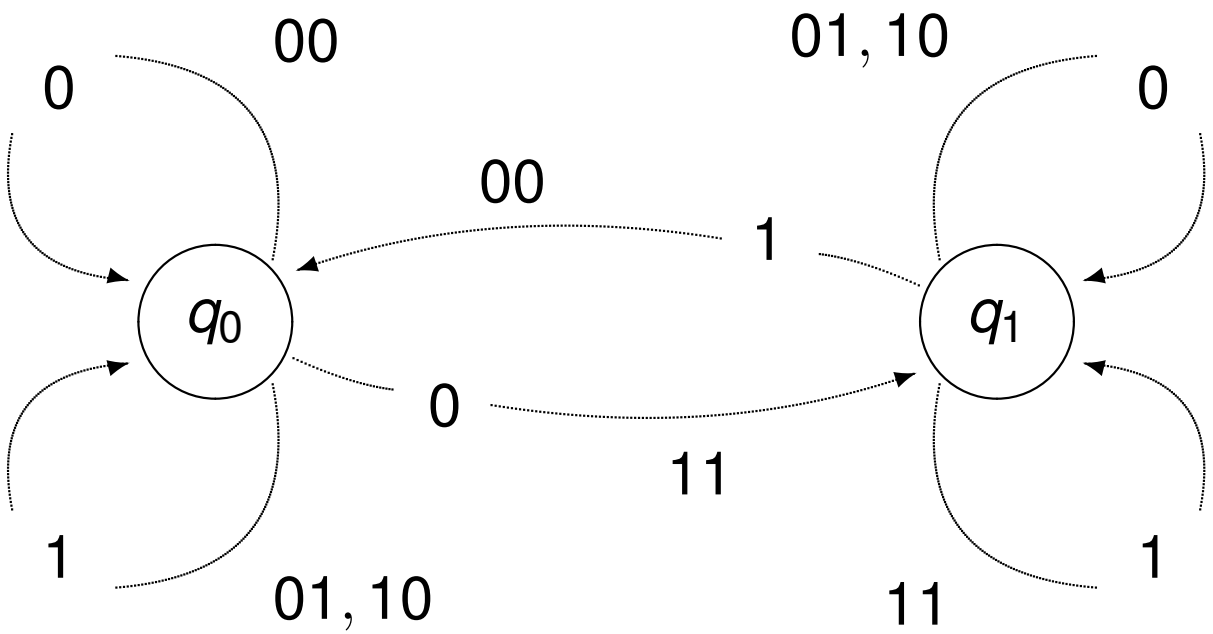
- Let U be a class of functions which is closed under composition, i.e., if $f, g \in U$, then $f \circ g \in U$. Observe that the composition of functions is associative.
- A Mealey machine is a deterministic finite automaton, which produces an output whenever reading a letter.
  - A Mealey machine $\mathcal{M}$ with state set Q, program δ and alphabet Σ defines for each letter $a \in \Sigma$ the function

$$f_a : Q \to Q \text{ with } f_a(q) := \delta(q, a)$$

  - For initial state q_0 and input word $w = w_1 w_2 \dots w_n$, we obtain the sequence $q_0, f_{w_1}(q_0), f_{w_2} \circ f_{w_1}(q_0), , \dots f_{w_n} \circ \dots \circ f_{w_2} \circ f_{w_1}(q_0)$ of states.
  - We can determine the sequence of states with the prefix problem for $U = \{ f \mid f : Q \to Q \}$ and $*$ as the composition of functions.

Addition as a Prefix Problem

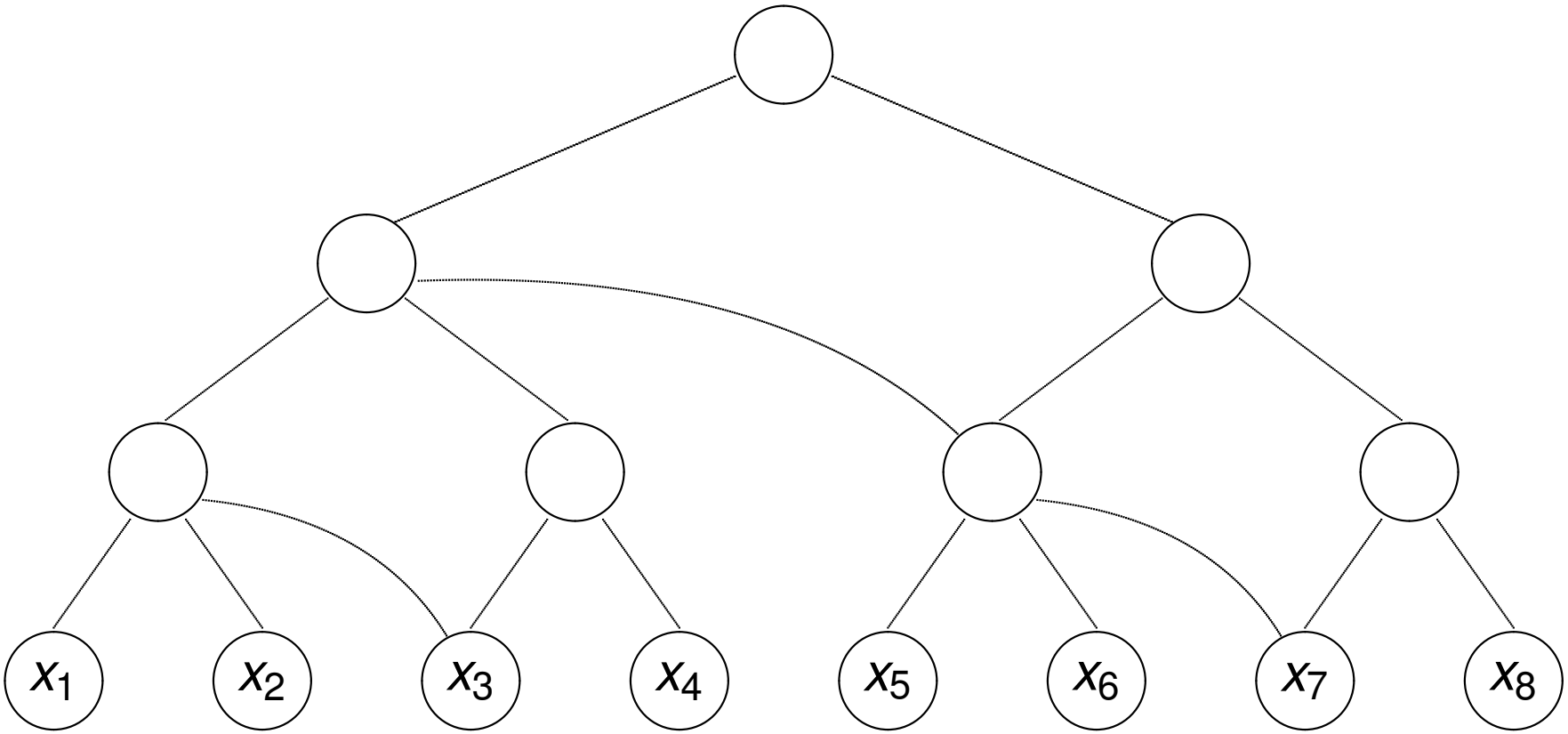
Adding two n-bit numbers $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_n$ becomes a prefix problem. Why? Consider the Mealey machine



(The last letter has to be 00.) The ith output bit is determined by x_{n-i+1}, y_{n-i+1} and the ith state in the sequence of states.

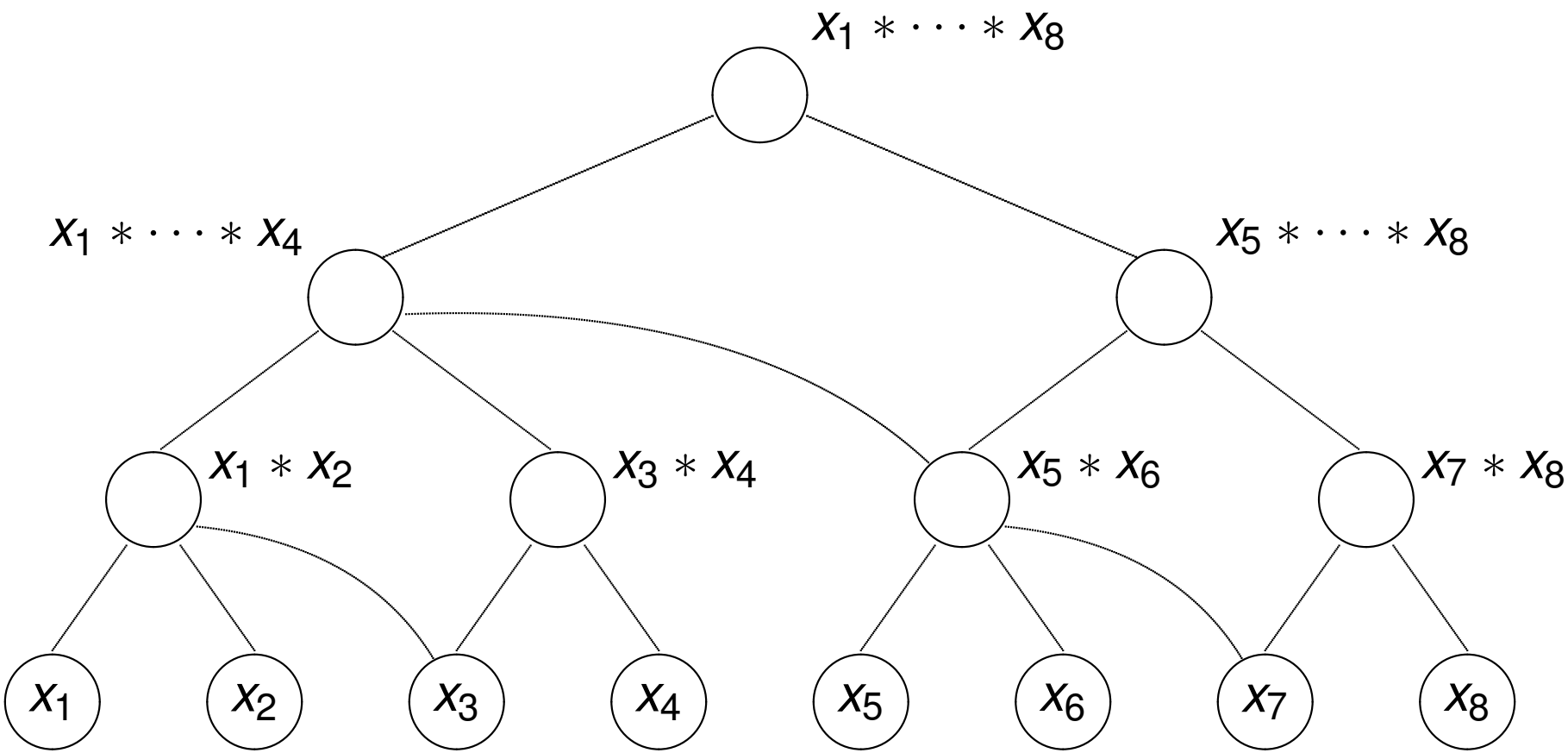
Solving the Prefix Problem

In T_k: introduce diagonal edges linking a left child to its leftmost nephew. Let T_k^* be the new communication pattern.



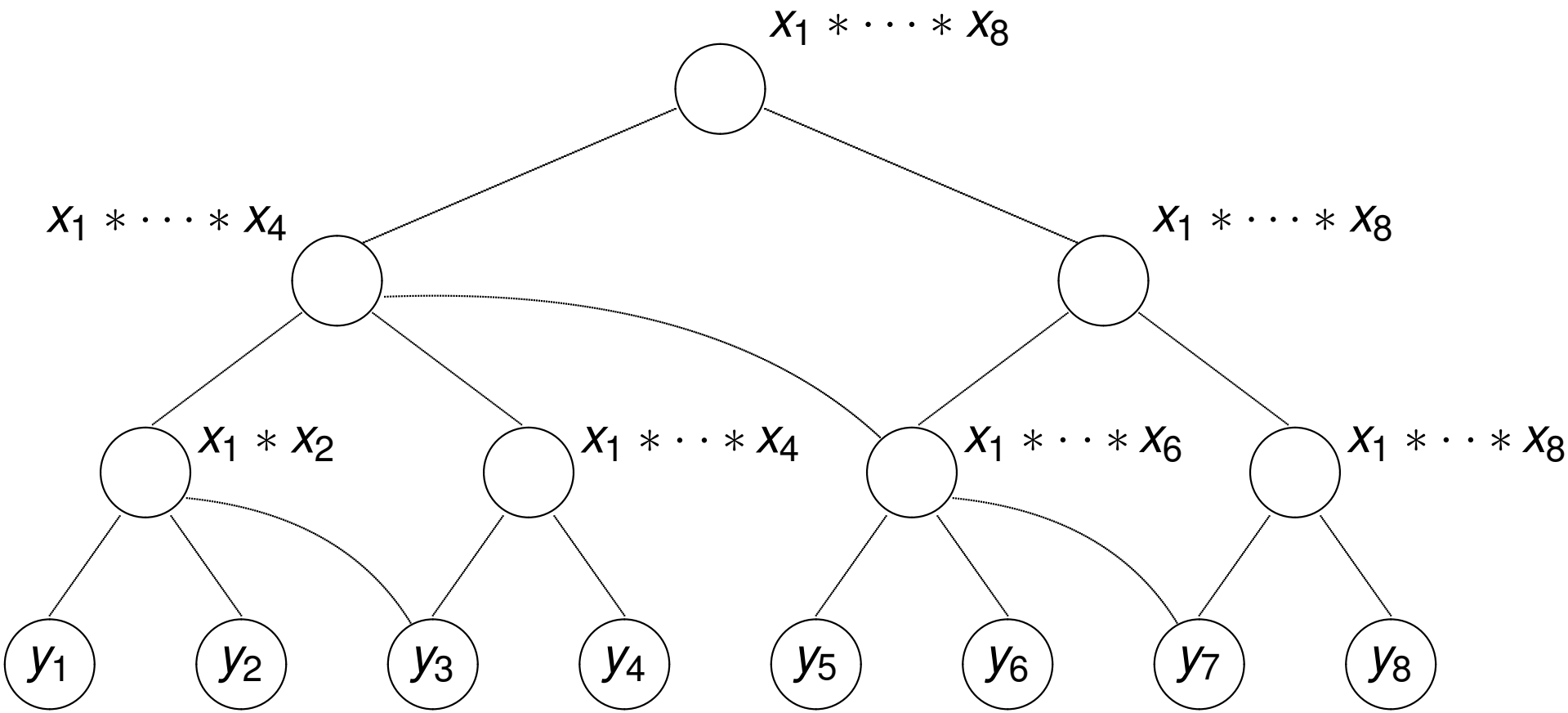
Solving the Prefix Problem: Phase 1

Assign input x_i to the ith leaf. In Phase 1 the input is moved up the tree and each interior node, upon receiving x and y from its children, computes $z = x * y$, stores the result z and hands z over to its parent.



Solving the Prefix Problem: Phase 2

The computation is moving down. Nodes receive information from their parent and have to pass it down to their children.



The Details of Phase 2

- The root wakes up its two children and sends its result to its right child.
  // The rightmost leaf will need the sum $x_1 * \cdot * x_n$.
- Assume that node v receives a wake-up call and that it currently stores the result z_v. Firstly v wakes up its two children.
  - If v is a left child, which does "sit" on the leftmost path, then it sends z_v to its nephew and its right child.
  - If v is a left child, which does not "sit" on the leftmost path, then it reads the data x it receives from its uncle. v sends x to its left child and $x * z_v$ to its nephew and its right child.
  - If v is a right child, then it reads the data x it receives from its parent, and sends x to its right child.

Performance Analysis

- What to do if we have to solve the prefix problem for $x_1, \ldots, x_n$ with 2^k processors?
- Each processor receives $n/2^k$ "numbers".
  - the processors solve their prefix problems in parallel in $O(n/2^k)$ compute steps.
  - the respective results are further processd in phase 1 and 2 for $O(k)$ compute and communication steps and
  - finally each processor determines its $n/2^k$ prefixes in time $O(n/2^k)$.

The prefix problem for n numbers can be solved with p processors in $O(\frac{n}{p} + \log_2 p)$ compute steps and in $O(\log_2 p)$ communication steps. In particular, two n-bit numbers can be added in time $O(\frac{n}{p} + \log_2 p)$ with $O(\log_2 p)$ communication steps.

The Mesh

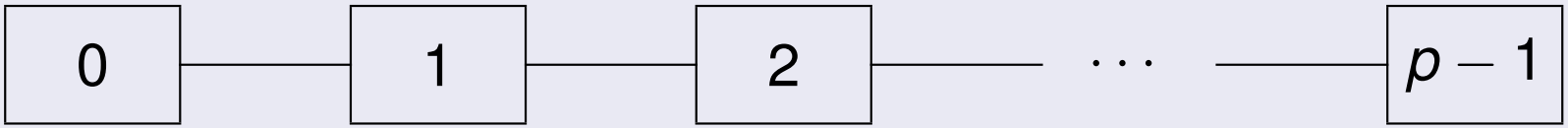
The d-dimensional mesh $M_d(m)$ is an undirected graph whose nodes are all m^d possible vectors $x = (x_1, \ldots, x_d)$ with $x_i \in \mathbb{Z}_m := \{0, 1, \ldots, m-1\}$.
Two nodes x and y are connected by an edge iff $\sum_{i=1}^{d} |x_i - y_i| = 1$.

- Nodes x and y are connected iff there is a coordinate i such that $y = (x_1, \ldots, x_{i-1}, x_i \pm 1, x_{i+1}, \ldots, x_d)$.
- $M_d(m)$ has degree at most $2d$ and diameter $d \cdot (m-1)$.
- The bisection width? Fix all but the first component per vector and remove all m^{d-1} edges incident with vectors $(m/2 - 1, x_2, \ldots, x_d)$ and $(m/2, x_2, \ldots, x_d)$.

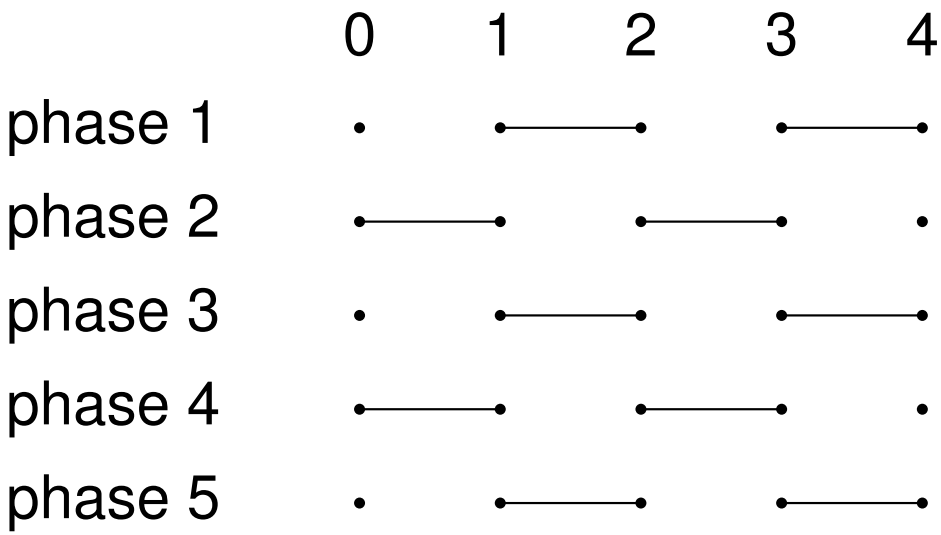
  We have disconnected the nodes into the two components $V_1 = \{(i, x_2, \ldots, x_d) \mid 0 \leq i < \frac{m}{2}, 0 \leq x_2, \ldots, x_d < m\}$ and $V_2 = \{(i, x_2, \ldots, x_d) \mid \frac{m}{2} \leq i < m, 0 \leq x_2, \ldots, x_d < m\}$.

Odd-Even Transposition Sort

Sort n numbers with the linear array $M_1(p)$.



We assume that p is odd and show how to parallelize Bubblesort.
If $n = p$, then we perform compare-and-exchange steps between "adjacent" processors.



Odd-Even Transposition Sort

// Each node receives n/p elements.

(1) Each node sorts its subsequence with Quicksort;

(2) for $k = 1$ to p do
    if (k is odd) then
        for $i = 1$ to $\lfloor \frac{p}{2} \rfloor$ pardo
            node $2 \cdot i$ sends its sorted sequence to $j = 2 \cdot i - 1$.
            j merges the two sorted sequences, keeps the smaller
            and returns the larger half to $2 \cdot i$;
    else
        for $i = 1$ to $\lfloor \frac{p}{2} \rfloor$ pardo
            node $2 \cdot i$ sends its sorted sequence to $j = 2 \cdot i + 1$.
            j merges the two sorted sequences, keeps the larger
            and returns the smaller half to $2 \cdot i$;

Analysis of the Odd-Even Transposition Sort

Odd-even transposition sort (OETS) sorts n keys on a linear array with p processors. OETS executes $O(\frac{n}{p} \cdot \log_2 \frac{n}{p} + n)$ compute steps and p communication steps involving messages of length n/p.
Its work ($=$ time $\times$ no. of processors) is at most $O(n \log \frac{n}{p} + n \cdot p)$.

- Resources: The initial sorting of n/p elements runs in time $O(\frac{n}{p} \log_2 \frac{n}{p})$. Each of the p phases runs in time $O(\frac{n}{p})$.
- Communicating subsequences dominates merging!
- Correctness: The crucial idea is the 0-1 principle.
  - A comparison-exchange algorithm uses only the compare-and-exchange operation.
  - A comparison-exchange algorithm is oblivious iff it applies the same operations for all input sequences.
  - Odd-even transposition sort is an oblivious comparison-exchange algorithm.

The 0-1 principle

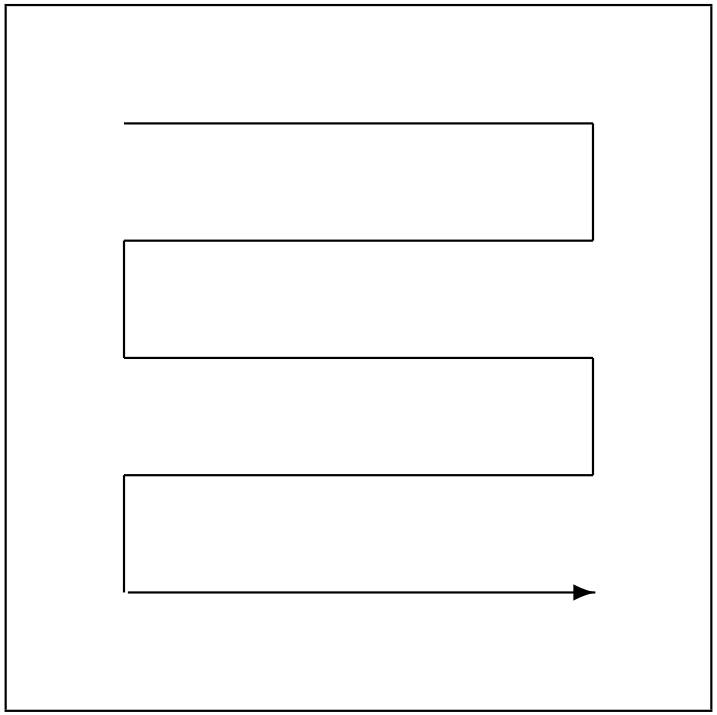
An oblivious comparison-exchange algorithm sorts correctly iff it sorts 0-1 sequences correctly.

- Assume 0-1 sequences are sorted, but the sequence $x = (x_1, \ldots, x_n)$ is incorrectly sorted.
- Let π be an order type of x, i.e., $x_{\pi(1)} \leq x_{\pi(2)} \leq \cdots \leq x_{\pi(n)}$. Assume that the wrong order type σ is determined.
- Let k be minimal with $x_{\pi(k)} < x_{\sigma(k)}$. Hence there is $l > k$ with $\sigma(l) = \pi(k)$.
  - Define the 0-1 sequence $y_i = \begin{cases} 0 & x_i \leq x_{\pi(k)}, \\ 1 & x_i > x_{\pi(k)}. \end{cases}$
  - Since $x_i \leq x_j \Rightarrow y_i \leq y_j$, y produces the same outcome of comparisons as x.
  - But 0-1 sequences are sorted and hence $y_{\sigma(1)} \leq \cdots \leq y_{\sigma(k)} \leq \cdots \leq y_{\sigma(l)} \cdots$.
  - However $y_{\sigma(k)} = 1$ and $y_{\sigma(l)} = 0$. Contradiction.

Correctness of OETS

- We apply the 0-1 principle.
- Let x be a sequence of n zeroes and ones.
  - With the possible exception of the first step, the n/p rightmost ones move right in every step until node $p-1$ is reached.
    (A rightmost one stays put, if it initially occupies an even node.)
  - Ones from the rightmost block are moving right after step 1.
- Inductive claim: ones from the ith rightmost block move right not later than step $i+1$.
  - Basis $i=1$ is correct.
  - Inductive step: ones from the $i+1$st rightmost block can be blocked only by ones from the jth rightmost block for $j \leq i$. But ones from the jth rightmost block move right not later than step $j+1 \leq i+1$.
  - The worst that can happen to a one from the $i+1$st block is to be blocked in the first i steps and to sit on a right node when comparing in step $i+1$.
- Ones from the ith rightmost block have to reach node $p-i$. The maximal distance $p-i$ can be traveled in steps $i+1, \ldots, p$.

Shearsort: Sorting on $M_2(n)$



- In odd phases sort all rows in parallel.
  - To sort odd rows all smaller keys move left, and
  - to sort even rows all smaller keys move right.
- in even phases sort all columns in parallel.

Apply odd-even transposition sort in each sorting step.

Shearsort is done after $2 \log_2 n$ phases

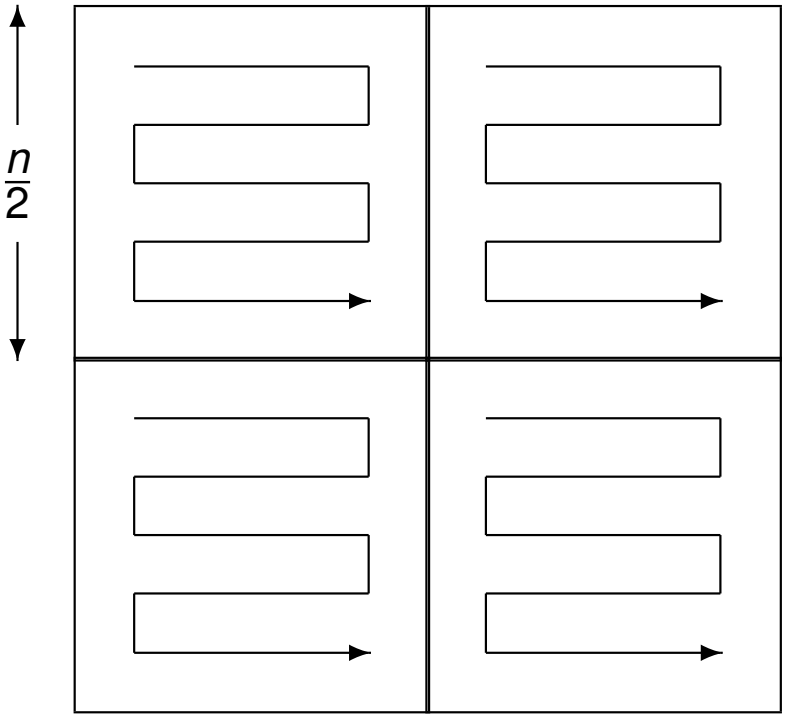
- Assume that rows $2i - 1$ and $2i$ are snake-sorted with odd-even transposition sort.
  Call a row clean, if it has only zeroes or only ones.
- **Case 1:** the number of zeroes of row $2i$ is at least as large as the number of ones of row $2i - 1$.
  - Apply the very first step of column sorting.
  - Row $2i - 1$ turns clean and moves up during column sorting.
- **Case 2:** the number of zeroes of row $2i$ is less than the number of ones of row $2i - 1$.
  - Apply the very first step of column sorting.
  - Row $2i$ turns clean and moves down during column sorting.
- With an inductive argument: after phase k 0-rows are followed by at most $n/2^k$ dirty rows and finally only 1-rows appear.

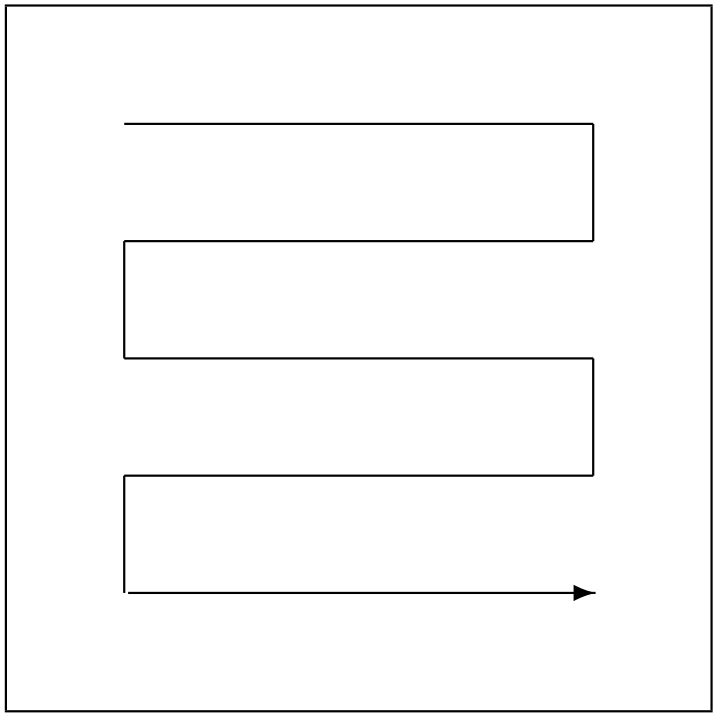
Shearsort with p processors

- Imagine p processors arranged in a $\sqrt{p} \times \sqrt{p}$ mesh of processors.
- Each processor receives $\frac{n}{p}$ keys.
- Each processor sorts its n/p keys in time $O(\frac{n}{p} \cdot \log_2 \frac{n}{p})$ with, say, quicksort.
- Any application of odd-even transposition sort involves $\sqrt{p}$ processors and hence runs in time $O(\sqrt{p} \cdot \frac{n}{p}) = O(\frac{n}{\sqrt{p}})$.

Shearsort sorts n keys in time $O(\frac{n}{p} \cdot \log_2 \frac{n}{p} + \frac{n}{\sqrt{p}} \cdot \log_2 p)$ on $M_2(\sqrt{p})$. Thus its work is bounded by $O(n \cdot \log_2 \frac{n}{p} + n \cdot \sqrt{p} \cdot \log_2 p)$. It uses $O(\sqrt{p} \cdot \log_2 p)$ point-to-point communication steps involving $\frac{n}{p}$ keys.

An improvement





- Sort n^2 keys recursively on the two-dimensional mesh M_n: recursively sort the four quadrants of $M_{n/2}$ in snakelike order.
- Sort the rows in snakelike order and then the columns.
- Finally sort the "snake" with $2 \cdot n$ steps of OETS.

Performance Analysis

- Without proof: The recursive sorting procedure is correct.
- If $T(n)$ is the running time for n^2 processors when sorting n^2 keys, then
$$T(n) = T(\frac{n}{2}) + O(n).$$
- Running time $T(n) = \Theta(n)$ suffices, compared to running time $\Theta(n \cdot \log_2 n)$ for Shearsort.
- If N keys are sorted with p processors, then the running time is bounded by
$$O(\frac{N}{p} \log_2 \frac{N}{p} + \sqrt{p} \cdot \frac{N}{p}).$$
- Its work is bounded by $O(N \cdot \log_2 \frac{N}{p} + \sqrt{p} \cdot N)$, compared to $O(N \cdot \log_2 \frac{N}{p} + \sqrt{p} \cdot N \cdot \log_2 p)$ for Shearsort.

The Hypercube

The d-dimensional hypercube $Q_d = (V_d, E_d)$ has node set $V_d = \{0,1\}^d$. The edges of Q_d are partitioned into the sets

$$\{ \{w, w \oplus e_i\} \mid w \in \{0,1\}^d \}$$

of edges of dimension i (for $i = 1, \ldots, d$).
e_i has a one in position i and zeroes elsewhere.

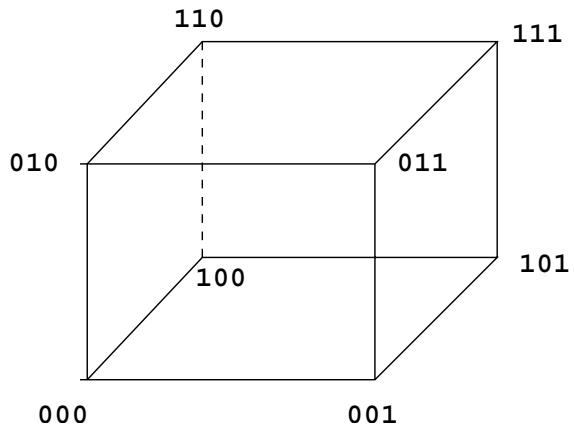
- The hypercube is a mesh, since $M_d(2) = Q_d$.
- Nodes of Q_d are binary strings $u \in \{0,1\}^d$ of length d.
- Nodes u, v are connected by an edge iff v can be obtained from u by flipping exactly one bit.
  Edges of dimension i correspond to flipping the ith bit.
  - Q_d has degree d, since any node has exactly d neighbors, one neighbor per dimension.
  - Q_d has diameter d, since at most d bits have to be "fixed".

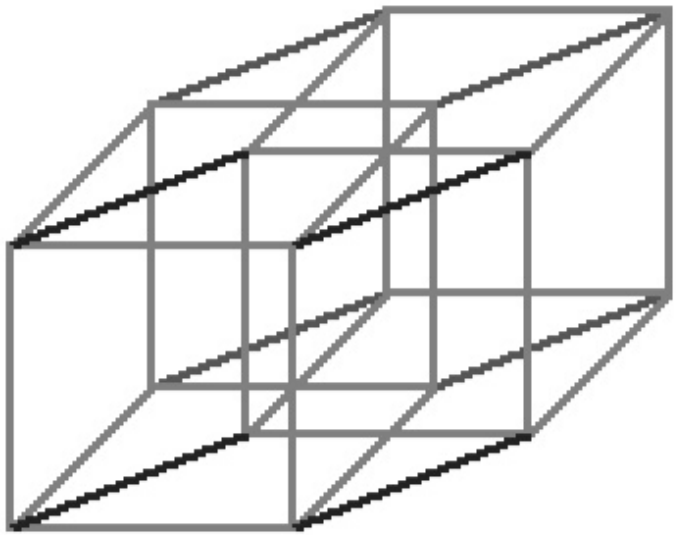
The Recursive Structure of the Hypercube

Q_d consists of two copies of Q_{d-1}:

- the 0-hypercube of all nodes $0x$ and
- the 1-hypercube of all nodes $1x$.
- Edges connect node $0x$ and node $1x$.

The 3- and 4-dimensional hypercube:





Bisection Width and Robustness of the Hypercube

- Consider a subset S of nodes of the d-dimensional hypercube.
- The cut $C(S, \bar{S})$ between S and $\bar{S}$ is the set of edges whose removal disconnects S and $\bar{S}$.

For every subset of nodes S with $|S| \leq |\bar{S}|$:

$$|C(S, \bar{S})| \geq |S|.$$

- To disconnect S from the rest of the hypercube, at least $|S|$ edges have to be removed, provided $|S| \leq |\bar{S}|$.
- The bisection width of Q_d is exactly 2^{d-1}:
  - The bisection width is the minimal number of edges whose removal disconnects half of the nodes from the rest.
  - Call the "first" half S and $|S| = |\bar{S}| = 2^{d-1}$ follows.
  - Hence at least 2^{d-1} edges have to be removed.

Disconnecting S

By induction on d show $C(S, \bar{S}) \geq |S|$, provided $|S| \leq |\bar{S}|$.

- $d = 1$ is trivial.
- Let $S \subseteq V_d$ be arbitrary with $|S| \leq |\bar{S}|$. Hence, $|S| \leq 2^{d-1}$.
- $S_0 \subseteq S$ (resp. $S_1 \subseteq S$) is the set of nodes of S in the 0-hypercube and 1-hypercube respectively.
  - If $|S_0| \leq 2^{d-1}/2$ and $|S_1| \leq 2^{d-1}/2$: Apply induction to each subcube and $C(S, \bar{S}) \geq |S_0| + |S_1| = |S|$.
  - Suppose $|S_0| > 2^{d-1}/2$. Then $|S_1| \leq 2^{d-1}/2$.
    - ★ By induction there are at least $|\bar{S_0}| = 2^{d-1} - |S_0|$ edges of $C(S, \bar{S})$ within the 0-subcube and at least $|S_1|$ edges within the 1-subcube.
    - ★ At least $|S_0| - |S_1|$ edges in $C(S, \bar{S})$ connect the two subcubes: at most $|S_1|$ edges connect nodes in S_1 with nodes in S_0.
    - ★ The cut consists of at least $(2^{d-1} - |S_0|) + |S_1| + (|S_0| - |S_1|) = 2^{d-1}$ edges and we are done since $|S| \leq 2^{d-1}$.

Why is the hypercube a successful communication pattern?

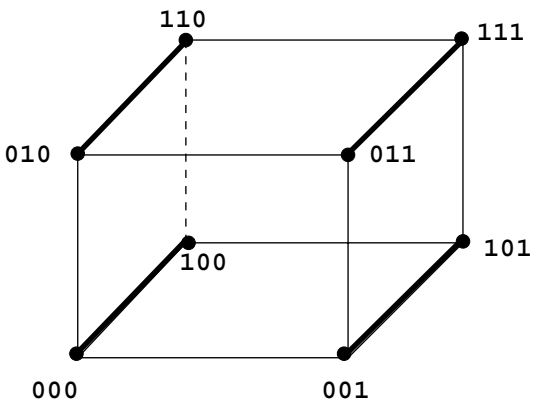
- The diameter of Q_d is d and hence logarithmic in the number 2^d of nodes. Thus any two nodes can communicate after logarithmically many steps.
- The previous property also applies to the complete binary tree T_d. But the bisection width of T_d is one, whereas the bisection width of Q_d is $2^{d-1} \Rightarrow$ Q_d has no bottlenecks.
- A good network should be able able to simulate other important networks without significant congestion and delay: Q_d can simulate trees and meshes with ease.
- Q_d is tailor-made to simulate divide & conquer algorithms through its recursive structure.

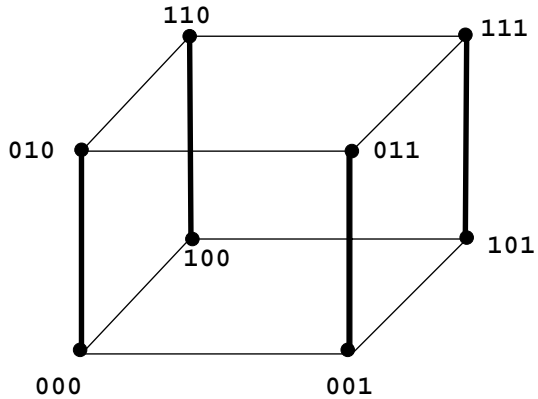
Simulating Trees

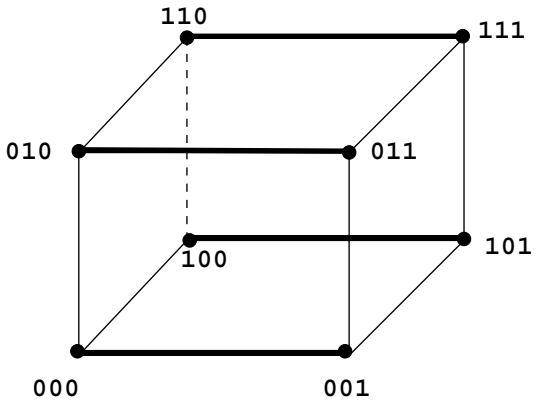
- Bad news: the complete binary tree T_d is not a subgraph of Q_{d+1}. Why?
  - Hypercube edges flip the parity of nodes.
  - All nodes of Q_{d+1}, which correspond to tree nodes of same depth, have the same parity.
  - $2^{d-2} + 2^d > 2^d$ hypercube nodes correspond to tree nodes of depth $d-2$ or d and these nodes have the same parity.
  - There are exactly 2^d hypercube nodes of parity zero respectively one in Q_{d+1}. Contradiction.
- However we can simulate normal tree algorithms easily by normal hypercube algorithms.

Normal Algorithms for Trees and Hypercubes

- A tree algorithm is a normal tree algorithm, iff
  - only nodes of identical depth are used at any time. Moreover,
  - if only nodes of depth i are used at time t, then only nodes of depth $i-1$ or only nodes of depth $i+1$ are used at time $t+1$.
- An algorithm for the hypercube Q_d is normal iff at any time t
  - Edges of only one dimension $d(t)$ are active, i.e., carry information.
  - At time $t+1$ only edges of dimension $d(t+1) = d(t) \pm 1$ (resp. $d(t+1) = 1$, if $d(t) = d$, or $d(t+1) = d$, if $d(t) = 1$) are used.

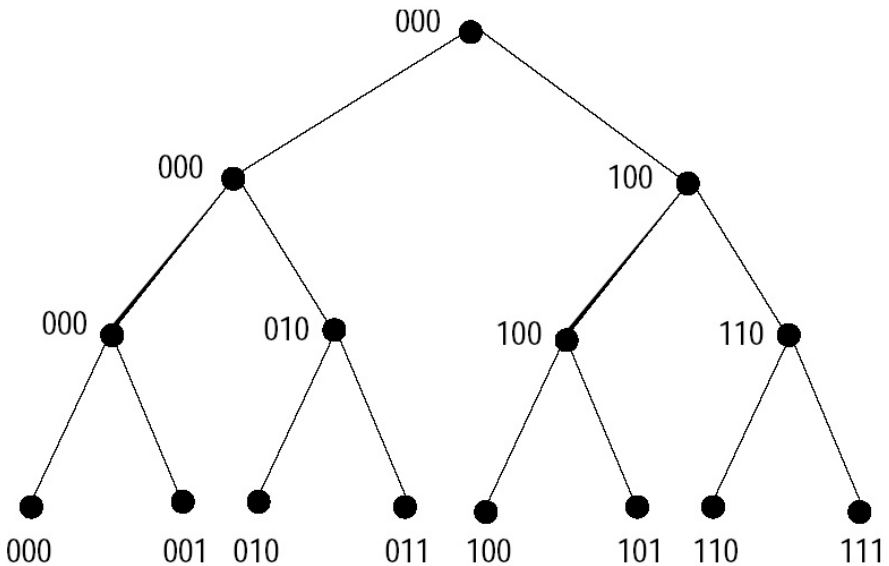






Simulating normal tree algorithms I

- Label an edge of T_d leading to a left child by zero and an edge leading to a right child by one.
- Identify the tree node v by label(v), where label(v) is the 0-1 sequence of labels on the path from the root to v.
- We simulate the tree node v by the hypercube node $v^* = \text{label}(\text{lm}(v))$, where lm($v$) is the leftmost leaf in the subtree with root v.



Simulating normal tree algorithms II

- Tree nodes of depth t are simulated by distinct hypercube nodes from the set $\{0,1\}^t \cdot 0^{d-t}$.
- Thus a communication with children is easy on hypercubes using the edges $\{w0^{d-t}, w10^{d-t-1}\}$:
  - a tree node and its left child are simulated by the same hypercube node.
  - Edges to right children, originating in tree nodes from the same layer, are mapped into hypercube edges of identical dimension.

Any normal tree algorithm on T_d can be simulated on the hypercube Q_d without any slowdown.

Simulating a Generalized Mesh I

The generalized mesh $M_d(n_1, \ldots, n_d)$, for numbers $n_1, \ldots, n_d \in \mathbb{N}$, is the undirected graph with node set

$$V = \times_{i=1}^{d} \{1, \ldots, n_i\}.$$

Two nodes u and v are connected iff $\sum_{i=1}^{d} |u_i - v_i| = 1$.

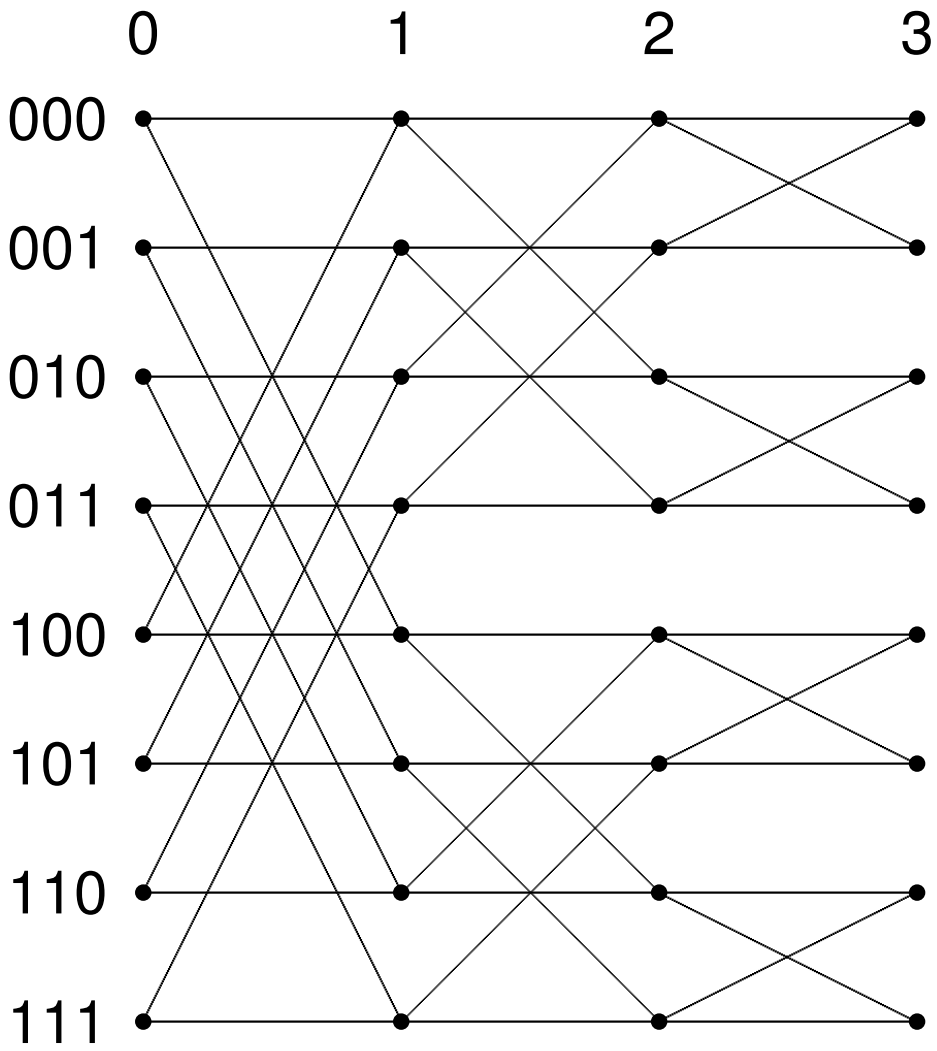
- The hypercube has a Hamiltonian path.
  - Proceed inductively, run the Hamiltonian path in subcube $0\{0,1\}^d$,
  - then "move up" into the subcube $1\{0,1\}^d$
  - and run the path backwards.
- Let P_j be a Hamiltonian path for the hypercube $Q_{\log_2 n_j}$.
  Map mesh node $(i_1, \ldots, i_d)$ to hypercube node $v^1(i_1) \circ \cdots \circ v^d(i_d)$:
  $v^j(i)$ is the ith node of P_j, and $\circ$ denotes concatenation of strings.

Simulating a Generalized Mesh II

- The mapping of mesh nodes to hypercube nodes is injective.
- Any edge of $M_d(n_1, \ldots, n_d)$ connects two nodes which differ in exactly one coordinate j and their difference is 1 or -1.
  - If the first endpoint is mapped to $v^1(i_1) \circ \cdots \circ v^j(i_j) \circ \cdots \circ v^d(i_d)$, then the second endpoint is mapped to (say) $v^1(i_1) \circ \cdots \circ v^j(i_j + 1) \circ \cdots \circ v^d(i_d)$.
  - The two hypercube nodes are connected, since $v^j(i_j)$ and $v^j(i_j + 1)$ are connected in $Q_{\log_2 n_j}$.

If $D = \sum_{i=1}^{d} \log_2 n_i$, then $M_d(n_1, \ldots, n_d)$ is a subgraph of Q_D.

The Butterfly Network B_3



The Butterfly Network

Q_d has the large degree d. The butterfly network B_d has degree 4 and can do almost what Q_d does.

- The nodes of the d-dimensional butterfly network B_d are pairs (u, i) with $u \in \{0,1\}^d$ and $0 \leq i \leq d$.
- Nodes (u, i) and $(v, i+1)$ are joined by a "horizontal edge" iff $u = v$ or by a "crossing edge" iff u and v differ only in their $(i+1)$st bit.
- We say that $C_i := \{(w, i) \mid w \in \{0,1\}^d\}$ is the $i$the column and $R_w := \{(w, i) \mid 0 \leq i \leq d\}$ is the row of $w \in \{0,1\}^d$.

B_d is tailor-made to simulate parallel computations of Q_d.

- The nodes of a row R_w play the role of node w of the hypercube.
- Hypercube edges of dimension i correspond to edges connecting nodes of C_{i-1} to nodes of C_i.

The Strengths of B_d

- The butterfly network B_d has $(d+1) \cdot 2^d$ nodes.
- Its degree is 4.
- The diameter of B_d is $2 \cdot d$.
- B_d has bisection width $\Theta(2^d)$.

B_d can simulate normal algorithms on the hypercube with slowdown 2.

Routing on the Hypercube

Implement the communication primitive "permutation routing":

- we are given a network $G = (V, E)$, a permutation $\pi : V \to V$.
- Each node $v \in V$ wants to send a message packet P_v along some path from v to node $\pi(v)$.
- We require that only one packet may traverse an edge at any time.
- Route all packets to their destination as quickly as possible.

- A routing algorithm is conservative, if the path of any packet depends only on the address $\pi(v)$ of the destination.
- Assume that G is an undirected graph with N nodes and degree D. If A is a conservative, deterministic routing algorithm, then there is a permutation π for which A runs for at least $\Omega(\frac{\sqrt{N}}{D})$ steps. Extremely slow!

Bit-Fixing for Q_d

- The bit-fixing strategy checks the address bits beginning with position 1 and ending with position d.
  - Assume that position i is considered and that packet P_v with address $\pi(v)$ has reached node w:
  - if w and $\pi(v)$ agree in their ith bit, then the packet does not move.
  - Otherwise, P_v is sent along the edge corresponding to bit i.
  - $(111, 011, 001, 000)$ is the bit-fixing path from $v = 111$ to $\pi(v) = 000$.
- Bit-fixing fails for the routing problem $(x, y) \rightarrow (y, x)$, where x (resp. y) is the first (resp. second) half of all bits of the sender v.
  - The packet of the node $(x, 0)$ wants to travel to $(0, x)$
  - and all $\sqrt{2^d}$ packets $(*, 0)$ visit the bottleneck $(0, 0)$.
  - Time $\Omega(\frac{\sqrt{2^d}}{d})$ is required.

Randomized Routing

- The maximal distance between any two nodes of Q_d is d. We should expect a good routing algorithm to run in time $O(d)$.
- Deterministic, conservative routing algorithms however require the unacceptable running time $\Omega(\sqrt{\frac{2^d}{d}})$ in the worst-case.
- We try a two-phase randomized routing approach:
  - First route all packets to a random destination using bit-fixing. (No bottlenecks to be expected.)
  - Then use bit-fixing to reach the original destination. (Again, no bottlenecks to be expected. Why?)

The Analysis: Two Basic Observations

- If two packets P_v and P_w meet at some time and diverge at some later time, then their paths will be node-disjoint from that point on.
- Assume that packet P_v runs along the path $(e_1, \ldots, e_k)$.
  - Let $\mathcal{P}$ be the set of packets which use some edge in $\{e_1, \ldots, e_k\}$ during phase one. Then the waiting time of packet P_v is at most $|\mathcal{P}|$.
  - Why? A packet P_w may repeatedly block a packet P_v!
  - Show the claim by induction on the size of $\mathcal{P}$. In particular: show that there is packet $P_w \in \mathcal{P}$, that blocks P_v at most once.

The Analysis

Introduce the random variables

$$H_{v,w} = \begin{cases} 1 & P_v \text{ and } P_w \text{ run over at least one joint edge,} \\ 0 & \text{otherwise.} \end{cases}$$

- The total waiting time of P_v during phase one is at most $\sum_w H_{v,w}$.
- How large is the expected value of $\sum_w H_{v,w}$ and how likely are large deviations?
- Let W_i be the number of bit-fixing paths which traverse the ith edge of P_v. Then $E[\sum_w H_{v,w}] \leq \frac{d}{2} \cdot E[W_1]$. Why?
  - Randomized routing: all edges carry the same expected load, namely the expected load $E[W_1]$ of the first edge of P_v.
  - The path travelled by P_v has expected path length $\frac{d}{2}$.
- $E[W_1] = 1$: the expected path length is $d/2$ and the load $d \cdot 2^d/2$ is distributed uniformly over all $d \cdot 2^d/2$ edges.

The Probability of Large Deviations

Chernoff: Sums of independent 0-1 valued random variables

$$\text{prob}[\sum_{w} H_{v,w} \geq (1+\beta) \cdot \frac{d}{2}] \leq e^{-\beta^2 \cdot \frac{d}{2}/3}.$$

- Set $\beta = 3$ and the probability of a total waiting time of at least $2 \cdot d$ is at most $e^{-3 \cdot d/2}$.
- The probability that some of the 2^d packets requires more than $2 \cdot d$ waiting steps is at most $2^d \cdot e^{-3 \cdot d/2} \leq e^{-d/2}$.
- The analysis of the second phase is completely analogous.

With probability at least $1 - 2 \cdot e^{-d/2}$, each packet

(a) reaches its destination in each phase after at most $3 \cdot d$ steps,

(b) respectively its final destination after at most $6 \cdot d$ steps.

Summary

- How to distribute information efficiently? Communication patterns are the data structures of parallel programming.
- Trees: Small diameter and small bandwidth.
  Fast solutions for easy problems such as prefix problems.
- Meshes: Relatively large bisection width already for small dimension.
  Ideal for problems on matrices.
- Hypercube architectures: Small diameter and large bisection width.
  - Ideal for parallelizing divide & conquer algorithms.
  - Support fast permutation routing.