Zaawansowane Techniki Algorytmiczne

Mirosław Kutyłowski 2016

Katedra Informatyki WPPT, PWr

Plan wykładu

1. modele obliczeń
   a. systemy równoległe
   b. systemy rozproszone
   c. sieci Boolowskie, OBDD
   d. obliczenia kwantowe i inne modele odbiegające od modelu von Neumanna
2. paradygmaty algorytmiczne
   a. samostabilizacja
   b. algorytmy aproksymacyjne
   c. algorytmy dla danych rozmytych
   d. algorytmy losowe
   e. derandomizacja
   f. algorytmy online
   g. uniwersalne heurystyki
3. analiza złożoności
   a. granice dolne
   b. złożoność komunikacyjna
   c. kombinatoryka analityczna
   d. rapid mixing
   e. rozwiązania dla ograniczonych zasobów

Cele

wykształcenie umiejętności w zakresie wykorzystania szerokiego spektrum zaawansowanych technik algorytmicznych, umiejętności z zakresie analizy poprawności i efektywności algorytmów

---

MODELS

I. Parallel computing

parallel computing today:

- supercomputers
- clasters of computers tightly connected (supercomputing centers)
- multicore architectures
- hardware like graphic cards
- some embedded systems

limitations for increasing computing power:

- each operation consumes energy, physical limitations
- increasing speed (frequency of the processor clock) increases energy consumption per second on a mm^2
- the same efect of more dense layout - large scale of integration
- ... but the heat has to removed in order to prevent overheating. This is hard (cooling systems)!
- $\Rightarrow$ progress on single processor architectures has been stopped after rapid progres during the 90's
- parallelisation as a remaining option

types of parallel machines:

- program execution:
  - SIMD - single instruction multiple data (all processors execute the same code and are in the same place in the program, typically used for „vector computations", that is matrices and so
  - MIMD - multiple instruction multiple data
- memory type:
  - shared (all locations accessible by all processors, problems of conflicts)
  - distributed (each processor has own memory, adressing remote memory via its owner)
  - hybrid (both types, shared memory slower and problematic, but sometimes extremely useful)

- interconnections:
  - mesh
  - hypercube or its variants (butterfly, deBruijn)
  - fat tree
- communication between the processors:
  - MPI (Message Passing Interface) a standard for point-to-point communication, buffers on endpoints, coordination: no message delivered before it is sent
  - shared memory (types: concurrent writes, collision, exclusive write,...)

- coordination between processors:
  - MPI: the programmer is fully responsible for it, logical structure of the algorithm has to ensure proper execution
  - shared memory: coordination from the global clock but ... latency, conflicts, ...
  - BSP (Bulk Synchronous Processing): supersteps, at the end of the superstep everything synchronised again, during the superstep the timing unpredictable
  - LogP: a model with synchronous clock and limitation of latency of communication
- parallelisation:
  - by hand: define processes (processes are assigned to processors by the manager of a parallel machine)
  - writing a code, giving to a compiler that recognizes parallelism and transfers into appropriate code
    - necessary to write programs with explicit parallelism in mind
    - in some languages explicit declarations

    generally: **a weak point (people think sequentially)**

application areas:

- numerical computations of linear algebra
- all spacial computations (2D, 3D,..) for instance in civil engineering, weather forecast,

- crypto - code breaking via brute force and many search algorithms
- bioinformatics, genetics, chemistry (simulating as a cheap and fast initial stage of design to eliminate most of wrng directions)
- modelling complex systems

computational complexity for parallel computing:

- **time** (sometimes called *depth*) from the start to the moment when the output is ready
  - if there is no fixed termination time it might be problematic to reach consensus when the computation has been finished
- **work** = the total amount of steps executed by all processors involved
  - generally the work cannot be lower than in case of a sequential execution (a sequential machine can always simulate a parallel computation)
  - typically the price for time speed-up is an increase of work

main problems:

- tme to market
- correctness of algorithms (practically infinite number of options for timing options of interprocessor communication)
- most programmers cannot think in terms of parallel programs
- some problems cannot be solved faster by parallel machines

Example of inherently parallel task:

compute H_n for a given I, t and n, where $H_0 = I$, $H_{i+1} = \mathrm{Hash}(H_i, i, t)$
Hash is a crytographic hash function

Knowledge assumption (simplified version): if a system learns h and y such that $h = \mathrm{Hash}(y)$ then with very high probability y must appear during the computation **before** h

Many algorithmic problems are hard to parallelize. It is an open theoretical question whether, say, all problems solvable in polynomial time can be executed in much shorter time by a parallel machine.

Some issues:

→ **load balancing** (utilize processors evenly):

- sophisticated approaches or

- random strategies (e.g. choose two processors at random and assign the task to a processor with less load – so called „power of two choices")

→ **symmetry breaking**: if no IDs assigned to processes then processes might be in exactly the same state and who should take which role. Solution: based on random choice

→ **concensus:** processes might have a different understanding of the global state

→ **Byzentine agreement:** the messages might be undelivered. What to do in this case? A majority $2/3$ of nonmalicious processors should be able to make a decision

(Byzantine agreement problem: there are two Byzantine armies. If they attack the enemy at the same time, then they win. Communication between the armies is via messengers that go through the enemy territory. The messangers can be captured - the sender cannot be sure that the message arrived at the destination)

ALGORITHM EXAMPLES

(borrowed from Parallel Algorithms by G.Blelloch and B. Maggs)

SUM: compute the sum elements in an array (shared memory)

- recursive call to the procedure for elements $A[2i]+A[2i+1]$ for $i=1,\ldots,|A|/2$
- work $O(n)$, time $O(\log n)$, nmber of processors can be $n/\log n$ (combination of sequential and parallel)

PARALLEL PREFIX: compute $\sum_{i=1}^{j} A[j]$, simultaneously for $j=1,2,\ldots.,n$

`ALGORITHM:` ParallelPrefix(A)
1 if $|A|=1$ then return $A[0]$
2 else
3 $S=\text{ParallelPrefix}(\{A[2i]+A[2i+1]\}_{i=1,\ldots,|A|/2})$
4 $R[i]:=S[i/2]$ if i even, else $R[i]:=S[i-1/2]+A[i]$ for $i\le n$

array R is the output

Work: $W(n)=W(n/2)+O(n)$ so $W(n)=O(n)$
Time $D(n)=D(n/2)+O(1)$ so $D(n)=O(\log n)$

POINTER-JUMPING: given a directed acyclic graph (e.g. a tree), find the root for each vertex

`ALGORITHM:` pointer-jumping(P)
for j from 1 to $\lceil \log |P| \rceil$
$P := P[P[i]]$ for $i \leq |P|$

- at each iteration the poiter jumps forwards, the exception are the roots that point to themselves
- if the pointer already is to the root then the pointer does not change
- generally, the length of jumps double at each iteration, as the maximal path has length $|P|$, no more than $\log |P|$ iterations are needed

LIST-RANKING: given a list represented via pointers, find the distance of each vertex from the head of the list

idea: like pointer jumping, but keep counting the distance to the node shown by the pointer

`ALGORITHM`: list-ranking(P)
assign $V[i] = 1$ unless $P[i] = i$ (pointer to itself)
for j from 1 to $\lceil \log |P| \rceil$
$V[i] := V[i] + V[P[i]]$ for $i \leq |P|$
$P := P[P[i]]$ for $i \leq |P|$

V is the output

Work is $\Theta(n \log n)$. Bad!

a typical improvement via random sampling:

- choose $n/\log n$ *start nodes* at random
- from each start node walk (1 process per start node, the walk is sequential) until another start node is encountered
- with high probability each walk stops after $O(\log n)$ steps
- solve the LIST-RANKING problem with the list of start nodes and initialized not with 1 but the distance to the next start node
- for the reduced problem the previous algorithm can be applied, it requires $O(n/\log n \cdot \log(n/\log n)) = O(n)$ work, – so within the bound $O(n)$
- walk back (in parallel starting from each starting node) and compute the distances on the way

execution time remains logarithmic, but the work is time$\cdot$ number of processsors $= O(\log n \cdot n / \log n) = O(\log n)$

REMOVING DUPLICATES: array contains entries, some of them appear more then once. Remove duplicates (leave only one position for a given value)

- if the range of the elements is small, the problem is easy to solve:
  - (in parallel) read a position in the input array,
  - if z found then write z into the output array $R[z] := z$

  concurrent write is necessary

a solution based on hashing:

ALGORITHM: remove-duplicates(V)
1 choose a prime m higher than $2 \cdot |V|$
2 fill TABLE with -1
3 $i := 0$
4 $R := \{\}$
5 while $|V| > 0$
6 in TABLE insert value j in position hash$(V[j], m, i)$ for each j
7 WINNERS := $\{V[j]$: TABLE(hash$(V[j], m, i) = j)\}$
8 append R with the list of winners
9 in TABLE insert value hash(k, m, i) in position k for each k in WINNERS h
10 leave in V only those k, for which TABLE[hash$(k, m, i) \neq k$]
11 $i := i + 1$

RESULT is the output

- fine tuning regarding the choice of m at each stage
- appending the list of winners requires parallel prefix
- too small m means a lot of collisions
- each stage uses a different i, so hash values are unrelated

SORTING

- QUICKSORT is an inherently parallel algorithm
- but RADIXSORT works also fine (requires that the valus are b bit numbers, time $O(b \cdot \log n)$, work $O(b \cdot n)$)

ALGORITHM: radixsort(A, b)

for $i = 0, \ldots, b - 1$
flags $:= \{(a \gg i) \bmod 2 \colon a \in A\}$
notflags $:= 1 -$ flags
$R_0 :=$ parallelprefix(notflags)
$s_0 :=$ sum(notflags)
$R_1 :=$ paralleprefix(flags)
$R[j] := R_0[j]$ if flags$[j] = 0$, else $R[j] := R_1[j] + s_0$ (computing ranks)
rewrite A: value $A[j]$ moved to position $R[j]$

A is the output

- stable sorting (order of the same elements preserved)
- first reorder according to the least significant bit, then 2nd, ...

BREADTH-FIRST-SEARCH:

`ALGORITHM`: BFS(s, G)
FRONT $:= [s]$
fill TREE with -1
TREE $[s] := s$
while |FRONT $\neq 0$|
$E :=$ flatten$(\{\{(u, v) \colon u\,\text{neighbor of}\,v\} \colon v \in \text{FRONT}\})$
$E' := \{(u, v) \in E \colon TREE[u] = -1\}$
append TREE with E'
FRONT $:= \{u \colon (u, c) \in E'\,\text{and}\,v = \text{TREE}[u]\}$

return TREE

E' created with concurrent write, possibly there are some loosers
the function flatten used due to different graph representations

CONNECTED-COMPONENTS: label all vertices in a component with the same label, different label for different connected components

- possible with BFS but inefficient
- solution based on graph contraction

`ALGORITHM`: randomcontraction(LABELS, E)
if $(|E| = 0$ then return LABELS
else
each vertex chooses a bit at random, a vertex called a child if 1 chosen
HOOKS $:= \{(u, v) \in E \colon \text{child}[u] = 1\,\text{and}\,\text{child}[v] = 0\}$

5 put in LABELS values from HOOKS
6 $E' := \{(\mathrm{LABELS}[u], \mathrm{LABELS}[v] \colon (u,v) \in E \,\mathrm{and}\, \mathrm{LABELS}[u] \neq \mathrm{LABELS}[v]\}$
7 $\mathrm{LABELS}' := \mathrm{randomcontraction}(\mathrm{LABELS}, E')$
8 insert to LABELS' entries $(u, \mathrm{LABELS}'[v])$ for $(u,v) \in \mathrm{HOOKS}$

LABELS' returned

ALGORITHM: deterministiccontract(labels, E)
1 if $(|E| = 0$ then return LABELS
2 else
3 $\mathrm{HOOKS} := \{(u,v) \in E \colon u > v\ \}$
4 put in LABELS values from HOOKS
6 with pointer jumping assign LABELS according to the roots of local trees
7 $E' := \{(\mathrm{LABELS}[u], \mathrm{LABELS}[v] \colon (u,v) \in E \,\mathrm{and\, labels\, different}\}$
8 return deterministiccontract(LABELS, E')

sometimes works badly:
a star graph with labels $1..n$ outside and label n in the middle of the star
– n recursive calls

MINIMUM_SPANNING_TREE: a graph where edges has weights,
look for a subgraph – a tree on all vertices and with a minimal weight

- easy: Lemma: for a node an outcoming edge of minimal weight belongs to the MST
- modify contract algorithm and a child always chooses an edge of the minimal weight (not a random one, not to a smaller node)

II. Distributed computing

(notices from the script of R. Wattenhofer, ETH Zurich)

Coloring algorithms

Quorum Systems

Quorum system idea:

- there are n servers, data is to be stored on all of them
- however, doing it at once is hard
- if we update a record on a subset of servers (a *quorum*) - we lock it and update
- in order to read a data the user checks all servers from some *quorum* and takes the mot recent one
- (we may assume that the data are authenticated together with time stamps so it is easy to see which version is the most recent one)
- different possibilities for quorums help as each server may fail to respond, ...

Problem:

- how to define quorums so that: any two quorums have non-empty intersection
- how to define access strategy: assign probability to each quorum and while reading choose a quorum according to this probability
- examples:
  - singelton: only one quorum consisting of one server
  - majority: every set of at least $n/2+1$ servers

Quality of a solution: **load**

- $L_Z(v)$ is load of a server v for a strategy Z: the probability that the server v will be read: $\sum_{v \in Q} \Pr(Z \text{ chooses } Q)$
- $L_Z(S)$ – the load of a strategy Z over a quorum S: max over all $L_Z(v)$
- $L(S) = L_Z(S)$ for the best strategy Z

Quality of a solution: **work**

- $W_Z(S)$ – work for strategy Z is the expected size of a quorum chosen
- $W(S)$ – work for the quorum S for the best strategy Z

Examples:
singelton: work=1, load=1

majority: work $>n/2$, load$\approx$0.5

Theorem $L(S) \geq 1/\sqrt{n}$

- consider a quorum Q with the smalles size q
- claim: for some $v \in Q$, $L_Z(v) \geq 1/q$, indeed: each time a quorum is accessed at least one server from Q is accessed as well, the probabilities of quorums sum up to 1 and they are „distributed" among q servers
- each time at least q servers are accessed, so there must be a server with $L_Z(v) \geq q/n$
- $L_Z(v) \geq \max(1/q, q/n)$. The best choice is $q = \sqrt{n}$

Grid quorum system:

- the servers form a grid $d \times d$ $(n = d^2)$
- each quorum is a set consisting of a column and a row
- each two quorums have 2 points of intersection
- option 1: a row and a column truncated below this row (only one intersection guaranteed)
- option 2: one row plus server per row in the rows below
- load $\approx 2/\sqrt{n}$

Locking problem:

- each access has to lock the quorum before writing (otherwise newer record might be overwrited by an older one)
- deadlock possible: e.g. in the grid system S_1 and S_2 intersect at s and s':
  S_1 locks s
  S_2 locks s'
  neither of them can proceed
- **Distributed Locking algorithm:**
  i. lock the nodes of a quorum one by one, according to their id numbers in an increasing way
  ii. if a locked server encountered then release all servers locked so far
- Claim: no deadlock possible
  Observation: the process that has locked the server with the highest id is either comple or can proceed (no node with a higher ID has been
- locked so far)

Fault tolerance

- up to f servers may fail, still there should be a quorum disjoint with the failed servers
- grid quorum system has f-resilience for $f < \sqrt{n}$, it is not $\sqrt{n}$ fault resilient (fail nodes on the diagonal)

Probabilistic failure

- each server works with pbb p
- What is the probability that a quorum system S fails? Notation: $F_p(S)$
- behavior: $F_p(S)$ inspected for big n:
  - for majority quorum system $F_p(S) \to 0$ for $p < 0.5$
    (follows from Chernoff Bound:
    for m independent binary variables x_i and success pbb p

$$\Pr\left(\sum_{i=1}^{m} x_i < (1-\delta)m \cdot p\right) < e^{-m \cdot p \cdot \delta^2/2}$$

  - for grid system, $F_p(S) \to 1$
    - we need at least one failed node per row to fail
    - $F_p(S) = (1-p^d)^d > 1 - d \cdot p^d \to 1$ (as $(1+x)^n > 1 + x \cdot n$ for $x > -1$

B-Grid

- a grid with $\boldsymbol{n = d \cdot h \cdot r}$ nodes, $\boldsymbol{d}$ colums
- $\boldsymbol{h}$ bands, each consisting of $\boldsymbol{r}$ rows
- "minicolumn" is a column in a band
- quorum: a minicolumn in each band, and a band with an element in each minicolumn
- $\boldsymbol{F_p(S) \to 0}$
  - failure if in each band a complete minicolumn fails or in one band in each minicolumn an element fails
    -
  - $\boldsymbol{F_p(S) \leq (d(1-p)^r)^h + h(1-p^r)^d}$
  - use $\boldsymbol{d = \sqrt{n}}$, $\boldsymbol{r = \ln(d)}$, $\boldsymbol{p \geq 2/3}$
  - so $\boldsymbol{(d(1-p)^r)^h \leq (d(1/3)^r)^h = d\left(d^{\ln 1/3}\right)^h \approx d^{\{-0.1h\}} < 1/n}$
  - so $\boldsymbol{h(1-p^r)^d < d(1-p^r)^d < d\left(1 - d^{\ln 2/3}\right)^d \approx d(1-d^{-0.4})^d \approx d \cdot e^{-d^{0.6}} = d^{-d^{0.6}/\ln d - 1} \ll d^{-2} \approx 1/n}$

Byzantine systems

- up to $\boldsymbol{f}$ servers may cheat

- $\boldsymbol{f}$-disseminating: every intersection of quorums contains at least $\boldsymbol{f+1}$ servers, there is a quorum without byzantine nodes
- so always at least one quorum survives (the Byzantine ones may pretend to crash)
- after writing in one quorum and reading by another one there will be a witness of the correct value (the Byzantine nodes may store an old/wrong value)
- enough if data authenticated
- as in the proof of the theorem above:
  $\boldsymbol{L(S) \geq \sqrt{(f+1)/n}}$
- $\boldsymbol{f}$-masking grid system: a quorum is a column and $\boldsymbol{f+1}$ rows, required: $\boldsymbol{2f+1 \leq \sqrt{n}}$
- M-Grid: $\boldsymbol{\sqrt{f+1}}$ rows and $\boldsymbol{\sqrt{f+1}}$ columns **,** quorums intersection have $\boldsymbol{2\sqrt{f+1}^2 = 2(f+2)}$ nodes

Opaque systems

- assume there is no data authentication
- two quorums intersect: Q_1 is up-to-date, Q_2 is not
- we require that the number of nodes in the intersection $Q_1 \cap Q_2 \setminus F$ is bigger than the number of byzantine nodes in Q_2 plus $Q_2 \setminus Q_1$
- would be great but...
  Theorem. For any f opaque system $L(S) \geq 0.5$
  Proof.
  i. size of $Q_1 \cap Q_2$ is at least half of the size of Q_1
  ii. load on Q_1:
  $$\sum_{v \in Q_1} \sum_{v \in Q_i} P_Z(Q_i) = \sum_i \sum_{v \in (Q_i \cap Q_1)} P_Z(Q_i) \geq \sum_i (|Q_1|/2) \cdot P_Z(Q_i) = |Q_1|/2$$
  iii. now by pigeonhole principle: there must be some node in Q_1 with at least load 0.5

III. Beyond von Neumann machines

Quantum computing and Shor factorization algorithm

Problem and algebraic context:

- given an RSA number $n = p \cdot q$ for prime factors p and q of similar size, find p or q
- many modern crypto products are based on difficulty of this factorization problem. Many embedded products with RSA, no update possible
- it suffices to learn a nontrivial root of 1: a number r different from -1 and 1 such that $r^2 = 1 \bmod n$, indeed
  $r^2 - 1 = (r-1)(r+1) = 0 \bmod p \cdot q$
  that is
  p divides either $r-1$ or $r+1$
  if p divides $r-1$ then q cannot divide $r-1$ as $r-1 < n$
  then $\mathrm{GCD}(n, r-1)$ yields p
- if for a given $a < n$ we find s such that $a^s = 1$, then with probability 0.5 we get $a^{s/2}$ as a nontrivial root of 1
- so the problem is to find such an s - the order of a

Qubit

- instead of a bit with discrete states 0 and 1 we have a linear combination of basis vectors (say $|0\rangle$ and $|1\rangle$:
  $\alpha \cdot |0\rangle + \beta \cdot |1\rangle$
  with α , β complex numbers
- a measurement of $\alpha \cdot |0\rangle + \beta \cdot |1\rangle$ yields $|0\rangle$ with pbb $|\alpha|^2$ and $|1\rangle$ with pbb $|\beta|^2$ – this is quite annoying but ...
- moreover: reading changes the state to the state read. In fact, this is the core of Shor'a algorithm - a reading creates a change in a physical system that would be infeasible to compute on a classical computer
- instead of a single bit we may have strings of qubits, say of length l where $l > n$

Quantum operations and gates

- a quantum computer should perform some operations on qubits, technical realization is a challenge, but in theory possible

- we consider $l-$qubit numbers as representing numbers mod 2^l (well, this is fuzzy as each bit is fuzzy), in this way we a get quantum state for each $a<q=2^l$
- Hadamard transformation: an easy way to create a quantum state such that takes any value a (denoted $|a\rangle$) with the same probability
  (we are skipping details)
- Quantum Fourier transform:
  - regular one: $(x_1,...,x_N)$ transformed to $(y_1,....,y_N)$ where
    $y_k=\frac{1}{\sqrt{N}}\sum_{j=0}^{N-1} x_j\cdot e^{(2\pi i\cdot j\cdot k)/N}$
  - quantum:
    $\sum x_i\cdot|i\rangle$ transformed to $\sum y_i\cdot|i\rangle$ where
    $y_k=\frac{1}{\sqrt{N}}\sum_{j=0}^{N-1} x_j\cdot e^{(2\pi i\cdot j\cdot k)/N}$
  - in other words:

    $$|j\rangle\rightarrow\frac{1}{\sqrt{N}}\sum_{k=0}^{N-1} e^{(2\pi i\cdot j\cdot k)/N}\cdot|k\rangle$$

  „efficient implementation" based on similar algebra as for DFT

Shor's algorithm (based on presentation of Eric Moorhouse)

1. fix q such that $2n^2<q<3n^2$, $q=2^l$ (or a product of small primes)
   we use states with $2l$ qubits, notation $|a,b\rangle|$ or $|a\rangle|b\rangle$
2. prepare state $|0,\ 0\rangle|$ and apply Hadamard transformation to the first register. result is a state
   $$|\psi\rangle=\frac{1}{\sqrt{q}}\cdot\sum_{a=0}^{q-1}|a,0\rangle$$
3. fix $x<n$ at random
4. apply to the state $|\psi\rangle$ the quantum transformation
   $$|a,0\rangle\rightarrow|a,x^a \bmod n\rangle$$
   the result is
   $$\frac{1}{\sqrt{q}}\cdot\sum_{a=0}^{q-1}|a,x^a \bmod n\rangle$$
   (there is a theory how to make such a computation with quantum gates)
5. measure the second register. The result is some k. But then the measured state changes to
   $$\frac{1}{\sqrt{M}}\cdot\sum_{d=0}^{M-1}|a_0+d\cdot r,k\rangle$$
   where A is the set of all a such that $x^a=k \bmod n$
   so $A=\{a_0,a_0+r,a_0+2r....\}$

and $M = |A|$ (so $M \approx q/r$)

6. apply the DFT to the first register. This changes the state

$$\frac{1}{\sqrt{M}} \cdot \sum_{d=0}^{M-1} |a_0 + d \cdot r, k\rangle$$

to

$$\frac{1}{\sqrt{q \cdot M}} \cdot \sum_{c=0}^{q-1} \sum_{d=0}^{M-1} e^{2\pi i \cdot c(a_0 + d \cdot r)/q} \cdot |c, k\rangle$$

which is equal to

$$\sum_{c=0}^{q-1} \frac{e^{2\pi i \cdot c \cdot a_0/q}}{\sqrt{q \cdot M}} \sum_{d=0}^{M-1} e^{2\pi i \cdot c \cdot d \cdot r/q} \cdot |c, k\rangle$$

$$\sum_{c=0}^{q-1} \frac{e^{2\pi i \cdot c \cdot a_0/q}}{\sqrt{q \cdot M}} \sum_{d=0}^{M-1} \zeta^d \cdot |c, k\rangle$$

gdzie

$$\zeta = e^{2\pi i \cdot c \cdot r/q}$$

7. measure the first register (this is the key moment!!)
   - which c is read depends on the values of $\sum_{d=0}^{M-1} \zeta^d$ – which corresponds to the probability
   - if $c \cdot r/q$ is not very close to an integer, then the the sum is $\frac{1 - \zeta^M}{1 - \zeta}$
   - if $c \cdot r/q$ is an integer, then we sum up M ones
   - so the former case is unlikely and the readings are concetntrated around values c such that

     $$c/q \approx d/r$$

     for an integer d
   - the rest is a classical computation involving c, q . The search space is relatively narrow

---

IV. Boolean circuits, decision diagrams

PARADYGMATY

ANALIZA