Zaawansowane Techniki Algorytmiczne

Mirosław Kutyłowski 2017

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, ...
    example: compting OR of n bits on a PRAM machine:
    - in time t each processor knows OR of $P(t)$ values
    - in time t each memory cell stores OR of $M(t)$ values
    - after writing: $M(t+1) = P(t) + M(t)$
    - after reading: $P(t+1) = M(t+1) + P(t)$
    - a Fibbonaci sequence $....M(t), P(t), M(t+1), P(t+1)....$
    - grows almost like exponentially, but not exactly
    - so a careful design which address to read and write
  - 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
    - L communication latency (time to deliver a message)
    - o - overhead (time needed to send or receive a message), at this time no other operation performed
    - g - gap, time between consecutive transmissions of messages
    - Example: computing $a_1 \times a_2 \times ... \times a_n$ for an associative operation $\times$:
      - obviously, a tree structure for collecting data and combining results

- we determine how big might be n given a time limit T:
  - → for $T \leq L + 2o$ communication would cost more, so compute everything locally
  - → otherwise the last step of the root is to combine its result and the result obtained from another processor, which has sent at time: $T - 1 - L - 2o$
  - → ... so we may get a recursive expression for the maximal n

- 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 wrong 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:

- time 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 non-parallel task:

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

Knowledge assumption (simplified version, one of fundamental assumtions for many cryptographic constructions):
if a system learns h and y such that $h = \text{Hash}(y)$, then with very high probability y must appear during the computation **before** h

Many algorithmic problems seem to be 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)

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

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

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)
1 for j from 1 to $\lceil \log |P| \rceil$
2 $P := P[P[i]]$ 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)
1 assign $V[i] = 1$ unless $P[i] = i$ (pointer to itself)
2 for j from 1 to $\lceil \log |P| \rceil$
3 $V[i] := V[i] + V[P[i]]$ for $i \leq |P|$

4 $P := P[P[i]]$ 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· number of processsors $= O(\log n \cdot n/\log n) = O(n)$

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 hash(k, m, i) in position k for each k from WINNERS

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 (trade-off between efficiency - lower m means faster appending - and probability of collisions -low m means a lot of collisions and many rounds
- 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)
1 for $i = 0, ..., b - 1$
2 flags $:= \{(a \gg i) \bmod 2 : a \in A\}$
3 notflags $:= 1 -$ flags
4 $R_0 :=$ parallelprefix(notflags)
5 $s_0 :=$ sum(notflags)
6 $R_1 :=$ palleprefix(flags)
7 $R[j] := R_0[j]$ if flags$[j] = 0$, else $R[j] := R_1[j] + s_0$ (computing ranks)
8 rewrite A: value $A[j]$ moved to position $R[j]$

A is the output

properties:

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

CONNECTED-COMPONENTS:
definition: two vertices in a graph are in the same connected component if there is a path from one vertex to the second vertex
goal: label all vertices in a component with the same label, different label for different connected components

- one may try BFS search but it is inherently sequential and inefficient in the parallel setting

– it is easier to solve the problem using graph contraction technique

`ALGORITHM`: randomcontraction(LABELS, E)
1 if ($|E| = 0$ then return LABELS
2 else
3 each vertex chooses a bit at random, a vertex called a child if 1 chosen
4 HOOKS := $\{(u, v) \in E$: child$[u] = 1$ and child$[v] = 0\}$
5 put in LABELS values from HOOKS (the child gets the label of the parent)
6 $E' := \{$(LABELS$[u]$, LABELS$[v]$: $(u, v) \in E$ and LABELS$[u] \neq$ LABELS$[v]\}$
7 LABELS$'$:= randomcontraction(LABELS, E')
8 insert to LABELS$'$ entries $(u,$ LABELS$'[v])$ for $(u, v) \in$ HOOKS

LABELS' returned

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

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

II. Distributed computing

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

below 2 interesting general topics from distributed systems. These are only to examples.

Problems:

– communication takes long time

- communication is unreliable: messages may disappear or come in a wrong order
- network nodes may fail or even be malicious
- no direct coordination, the processors may have inconsistent view of the computation as a whole

Client-server Systems

- multiple users
- multiple servers – executing the commands of the users
- each command form a user has to be executed on all servers
- problem with the order of receiving commands:
  - servers X, Y store $a = 0$
  - client A asks to compute $a := a + 1$
  - client B asks to compute $a := 2 \cdot a$
  - X receives order from A and then from B, result: $a = 2$
  - Y receives order from B and then from A, result: $a = 1$
  - inconsistency created!!

Serializer:

- a distinguished server that collects the commands from the clients and send them to the servers in the same order
- problem: serializer is a single point of failure
- but the goal of deploying multiple servers was to avoid problem if some server fails!

Locks:

- each client tries first to lock all servers and then execute the command
- no inconsistency
- but what if some server does not respond? the system is blocked
- some strategy to unlock the servers needed (when two clients lock successfully different servers at the same time)

Idea of tickets:

- tickets issued to clients by the servers

– the tickets need not to be returned
– server accepts the commands with the most recently issued tickets only

PAXOS:

client		**server**
initialization		
choose command c $t := 0$		$T_{\max} := 0$ $C = \perp$ $T_{\text{store}} := 0$
phase 1		
$t := t + 1$ ask each server for a ticket		

		if $t > T_{\max}$ then
		$T_{\max} := t$
		respond ok(T_{store}, C)
		end if

phase 2		

if a majority answers ok(...) then
pick $(T_{\text{store},}, C)$ with maximum T_{store}
if $T_{\text{store}} > 0$ then
$c := C$
send propose(t, c) to the servers		

		if $t := T_{\max}$ then $C := c$ $T_{\text{store}} := t$ respond `success`
phase 3		
if majority of answers `success` send `execute`(c) to every server		

properties:

– if $(t,\ c)$ accepted and stored by majority of servers, then later each `propose`(t', c') contains $c' = c$
  → take the smallest t^* for which this is not true
  → there must be a server s that has been envolved in both proposals
  → s received request for t^* after it has stored $(t,\ c)$ (otherwise t^* would not be a valid ticket)

→ but then the response would ok(t, c) and then $c' = c$

→ no way to get ok(t'', c') as t^* is the smallest one higher than t

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 most 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: $\sum_{v \in Q} \Pr(\text{strategy } Z \text{ chooses } Q)$
- $L_Z(S)$ – 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≈ 0.5

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

- consider a quorum Q with the smallest 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 expression on the right side is minimized for $q = \sqrt{n}$

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 a 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 completed 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

- with pbb p a server is not in the failure state
- 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 clever construction for low failure probability)

- 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, additionally: in one chosen band: 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 $\boldsymbol{d = \sqrt{n}}$, $\boldsymbol{r = \ln(d)}$, $\boldsymbol{p \geq 2/3}$
  - then $\boldsymbol{(d(1-p)^r)^h \leq (d(1/3)^r)^h = d(d^{\ln 1/3}) \approx d^{-0.1} < 1/n}$
  - then $\boldsymbol{h(1-p^r)^d < d(1-p^r)^d < d(1-d^{\ln 2/3})^d \approx d(1-d^{-0.4})^d \approx d \cdot e^{-d^{0.6}} = d^{(-d^{0.6}/\ln d)+1} \ll d^{-2} \approx 1/n}$

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)
- this is 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

– for any set F of f Byzantine nodes we require that the number of nodes in the intersection $Q_1 \cap Q_2 \setminus F$ is bigger than the number of nodes in $F \cap Q_2$ (cheating nodes) $\cup$ $Q_2 \setminus Q_1$ (old values)

– 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 because of the condition on opaque systems

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 its algebraic context:

- given an RSA number $n = p \cdot q$ for prime factors p and q of a similar size, the goal is to find p or q
- many modern crypto products are based on difficulty of this factorization problem. There are many software systems and embedded devices with RSA, no update is possible
- in order to break factorization problem it suffices to learn a nontrivial root r of 1:
  - $r \neq -1$
  - $r^2 = 1 \bmod n$

  indeed
  - $r^2 - 1 = (r-1)(r+1) = 0 \mod p \cdot q$
  - therefore p divides either $r-1$ or $r+1$
  - if p divides $r-1$ then q cannot divide $r-1$ as then $r-1$ would be at least n, but $r-1 < n$
  - in this situation we compute $\mathrm{GCD}(n, r-1)$, the result must be p
  - if p divides $r+1$ then q cannot divide $r+1$ and therefore q must divide $r-1$. In this case $\mathrm{GCD}(n, r-1)$ yields q
- 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. Indeed:
  - by Chinese Reminder Theorem a number $a < n$ is represented by $a_p = a \bmod p$ and $a_q = a \bmod q$
  - given a and b we may compute representation of $a \cdot b \bmod n$ by computing $a_p \cdot b_p \bmod p$ and $a_q \cdot b_q \bmod q$
  - there are two roots of 1 modulo prime number p: 1 and $p-1$
  - if $a^s = 1 \bmod n$, while $a^{s/2} \neq 1 \bmod n$, then $a^{s/2} \bmod p$ is 1 or -1
  - there are the following cases:
    1. $a^{s/2} = 1 \bmod p$, $a^{s/2} = -1 \bmod q$
    2. $a^{s/2} = -1 \bmod p$, $a^{s/2} = 1 \bmod q$
    3. $a^{s/2} = -1 \bmod p$, $a^{s/2} = -1 \bmod q$

the last case corresponds to $-1 \bmod n$, the first two ones to a nontrivial roots of -1

- so it suffices to find such an s - the order of a. By repeating the procedure for different a's we finally find a nontrivial root of $-1 \bmod n$

Qubit

the concept is as follows:

- instead of a bit with discrete states 0 and 1 we have a linear combination of basis vectors denoted by $|0\rangle$ 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: if the result is $|0\rangle$ then the physical state becomes $|0\rangle$ as well. There is no state $\alpha \cdot |0\rangle + \beta \cdot |1\rangle$ anymore.
- **In fact, this is the core of Shor'a algorithm - a reading operation 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 as a qubit), 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. The way to achieve this is:
  - create the state $|0....0\rangle$
  - apply Hadamard transformation gate to it
  - each ccordinate is transformed by

    so $|0\rangle$ is transformed to $\frac{1}{\sqrt{2}}\begin{pmatrix} 1, 1 \\ 1, -1 \end{pmatrix}$

    $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$

– Quantum Fourier transform:

- regular FT: $(x_1, ..., x_N)$ 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:

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

„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. Its 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. to the state $|\psi\rangle$ apply the quantum transformation
$$|a, 0\rangle \to |a, x^a \bmod n\rangle$$
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
$$\frac{1}{\sqrt{M}} \cdot \sum_{a \in A}^{M-1} |a, k\rangle$$
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 $M \approx q/r$)
$$\frac{1}{\sqrt{M}} \cdot \sum_{d=0}^{M-1} |a_0 + d \cdot r, k\rangle$$
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

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

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

where

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

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 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 concentrated 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

Classics about Boolean functions

- Boolean functions: fixed number of input varaibles, values 0 and 1
- representation:
  - formulae
  - DNF (construction: disjunction of all monomials where a monomial represents one point where the function has value 1)
  - CNF (e.g. construct DNF for negation and then negate DNF, use de Morgan principle)
  - the general problem is the number of function of n variables which is 2^{2^n} so by pigeonhole principle majority of them must have representation of the length 2^n

- length of the representation depends very much on the method of representation.
  example: $(x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge .... \wedge (x_n \vee y_n)$ (CNF is short), but DNF is disjunction of the monomials of the form

$$(z_1 \wedge z_2 \wedge .... \wedge z_n)$$

  where each z_i is either x_i or y_i

- Cook's Theorem:
  The following problem is NP-complete: given Booelan formula decide whether it is satisfiable
  (ϕ is satisfiable if there is an assignent a for variables such that $\phi(a) = 1$)
- for DNF formula satisfiability is easy, for CNF not
- a formula ϕ is a tautology if for every assignment a we have $\phi(a) = 1$. Deciding if ϕ is a tautology is easy for CNF, but uneasy for DNF

Decision diagrams

- INF operator if.. then .. else meaning $(x \wedge y_0) \vee (\neg x \wedge y_1)$

$$x \to y_0, y_1$$

- each function can be expressed with INF - Shannon principle:

$$f = (x_0 \wedge f_0) \vee (\neg x_0 \wedge f_1)$$

  where f_i is f where we assign value i for x_0
- formula with INF only: decision tree
- decision diagram: if two nodes have the same graph of successors then merge these nodes
- result: directed acyclic graph of outdegree 2, leaf nodes with values 0,1

Problems:

- for two decision diagrams decide whether they corrspond to the same function
- in particular: find out whether a diagram represents a tautology or is not satisfiable
- combine decision diagrams of f_1, f_2 for the function $f_1 \oplus f_2$ for a given operation $\oplus$

OBDD:

- solving the problems in general is impossible. So find a representation where it becomes easy

- there are no miracles: the price to be payed is perhaphs a long representation of a Boolean function
- OBDD conditions:
  - there is a fixed order of variables
  - in OBDD diagram the variables must be tested according to their order
  - example: OBDD for XOR
    [picture to come]
- construction of OBDD: follow the Shannon principle

ROBDD:

- reduced OBDD (ROBDD):
  - if for a node v both outcoming edges point to the same node, then v is eliminated
  - if v and u have the same graphs of successors, then merge v and u
- **Theorem:** for each function f and ordering of variables there is exactly one ROBBD for f
- **Proof** induction on the number of variables

Problem solving for ROBBD

- tautology: get ROBBD representation of f. If it is 1 then it is a tautology
- similar for satisfiability: RBDD representation is not 0
- so the procedure is:
  - find OBDD representation
  - perform reductions until no further steps possible
    (due to the Theorem we stop in the same representation no matter how we do it)
- problem: OBDD might have exponential size, so sometimes we fail

ROBDD and operations

- given ROBDD for f_1 and f_2 construct ROBDD for $f_1 \oplus f_2$
- example:
  many pictures to come

Applications

- example: 8 Queens Problem:
  1. define conditions as Boolean formulas

2. convert to ROBBD
3. check whether it is 0

– **model checking:**
1. discribe a system via Boolean variables and conditions
2. define transitions
3. derive conditions after transitions

– **CAD:** controlling the state of a system via conditions expressed as ROBDD's
the point is: relatively easy manipulations with operators

V. Randomized Algorithms

TBA (notes of Jacek Cichon)

1. Randomized: przykŞady: problem filozofów, testowanie równoąci wielomianów, prosty sposób wyboru lidera

2. Randomized: MinCut, nierównoą˘ Markowa, Czebyszewa, kolejne przykŞady, na koniec wpuącili mnie w dyskusjŚ o "Power of Two Choices" - powiedzieli mi, ťe mówiŞeą o tym, ale chceli jeszcze raz o tym porozmawia˘

3. Derandomizacja: MaxCut - metoda wartoąci oczekiwanej; wróciliąmy do MinCut, zauważyliąmy, ťe wystarczy niezaleťnoą˘ par, omówiliąmy jak z k bitów losowych moťna zrobi˘ 2^k-1 parami niezaleťnych zmiennych {0,1} (sumy po podzbiorach modulo 2) i jak to moťna wykorzysta˘.

ALGORITHMIC PARADIGMS

V. Online Algorithms

(based on Susane Albers lectures)

Setting and problem formulation

- the requests come online $\sigma_0, \sigma_1, ..., \sigma_n$ in a unpredictable way
- the request σ_i must be served immediately after it arrives
- the overall cost of the service has to be optimalized
- the problem: decision how to serve σ_i at the lowest cost depends on the future (unknown) requests
- the cost has to be compared with the optimal cost that occurs when we are given the whole sequence $\sigma_0, \sigma_1, ..., \sigma_n$ in advance

Paging Problem

- fast memory holds up to k pages
- in case of a memory request, the requested page is sought in the fast memory, if not found there („*page fault*"), then it is read from slow memory
- target: minimize the memory access time (= minimize the number of page faults)
- paradigm: leave in the fast memory the pages that will be used again
- problem: if we read in a page we have to evict one from the fast memory, **which one?**
- problem: we do not know which will be used again,

Optimal offline strategy (OPT):

- evict the page that will wait the longest time for its request

Competitive ratio

- given algorithm A, the cost of A (the number of page faults) is denoted by $C_A(\sigma)$
- target: find c and a such that

$$C_A(\sigma) \leq c \cdot C_{\mathrm{OPT}}(\sigma) + a$$

- if c is small we have a *guarantee* of quality
- A is called c-compatitive in this case

- warning: there might be A for which we cannot **prove** $C_A(\sigma) \leq c \cdot C_{\mathrm{OPT}}(\sigma) + a$ but nevertheless A behaves well

LRU Strategy

- least recently used page is evicted
- deterministic strategy
- LRU is k-compatitive:
  - assume that LRU and OPT start with the same pages in the fast memory
  - define epochs: LRU has exactly k faults on epoch $P(i)$ for $i > 1$ and at most k in epoch $P(0)$
  - let σ_{t_i} be the first request in $P(i)$ (so $\sigma_{t_{i+1}-1}$ is the last one)
  - let p be the last page requested in $P(i-1)$
  - **Claim:** $P(i)$ contains k requests to different pages , none of them to p
  - **Corollary:** OPT has at least one fault in $P(i)$: as it has p in the fast memory, it cannot hold other k pages from the claim
  - **Corollary:** $C_{\mathrm{LRU}}(\sigma) \leq c \cdot C_{\mathrm{OPT}}(\sigma) + k$
  - **Proof of the claim:**
    - claim holds if LRU has faults on different pages $\neq p$
    - assume that LRU has fault twice on q: $\sigma_{s_1} = q$, $\sigma_{s_2} = q$
      i. q is evicted at time t where $s_1 < t < s_2$
      ii. at this moment it is least recently used
      iii. so in time $s_1, ..., t$ there are requests to $k+1$ pages, so k of them $\neq p$
    - assume the LRU does not fault twice but one of the faults is p
      i. p generates a fault at time $t \geq t_i$
      ii. it must have been evicted by more recent pages in time $t_i, ..., t-1$
      iii. there must have been k other pages requested
- LRU is the best possible:
  `Theorem:` if a deterministic paging strategy is c compatitive, then $c \geq k$
  `Proof`
  for a deterministic strategy A and $k+1$ pages in total
  i. we choose a sequence of requests that each time the new request is the page that is not in the fast memory

ii. so A has 1 fault per request

iii. on a request: OPT evicts the page that will not be requested during the next $k-1$ steps

iv. so OPT has at most one fault per k requests

MARKING Algorithm

- surprisingly simple randomized algorithm,
- no immediate reason why it should be better than deterministic
- `Algorithm:`
  - initially all pages in the fast memory unmarked
  - when a page fetched it becomes marked, a randomly chosen unmarked page evicted
  - when all pages in the fast memory marked, then all marks removed and the game starts again
- quality measure: expected cost versus OPT
- `Theorem:` MARKING is $2H_k$-compatitive against any oblivious adversary that knows the requests in advance. That is

$$E[C_{\text{MARKING}}(\sigma)] \leq 2H_k \cdot C_{\text{OPT}}(\sigma)$$

($H_k = \sum_{i=1}^{k} \frac{1}{i} \approx \ln k$)

`Proof:`

i. divide the time into phases: during a phase plus the first step of the next phase: requests to exactly $k+1$ distinct pages, at the end of the phase all pages marked

ii. stale page = non marked page that has been marked during the previous phase

iii. clean page: neither marked nor stale

iv. goal to show: during a phase an amortized number of faults for OPT is at least $\frac{c}{2}$, while for MARKING the expected number is at most $c \cdot H_k$

v. analysis for OPT:

a) S_{OPT} the pages in the fast memory for OPT, S_M – for MARKINGS

b) $d_I = |S_{\text{OPT}} \setminus S_M|$ at the beginning of the phase

c) $d_F = |S_{\text{OPT}} \setminus S_M|$ at the end of the phase

d) let c be the number of clean pages requested during the phase

e) no clean page is in the memory of MARKINGS, $c - d_I$ clean pages are neither in S_M nor in S_{OPT}

f) so OPT has at least $c - d_I$ faults due to clean pages requests

g) at the end of the phase S_M contains only pages requested during the phase, d_F of these pages are missing in OPT, so they have been evicted due to some faults

h) the cost is $\geq \max(c - d_I, d_F)$

$$\max(c - d_I, d_F) \geq \frac{c - d_I + d_F}{2} = \frac{c}{2} - \frac{d_I}{2} + \frac{d_F}{2}$$

i) sum up over all phases:

$$\text{total cost} \geq (\text{number of all } c's) \cdot \frac{1}{2} - 0 + \frac{d_F}{2}$$

where $0 = \text{initial } d_I$

vi. analysis for MARKING for a phase:

a) serving c clean pages costs c (all faults)

b) there are at most $s = k - c$ stale requests (pessimistically we assume that $k - c$ are stale requests),

c) we have to compute the expected cost for the ith request to a stale page

d) $c(i)$ denotes the number of clean pages requested up to the ith stale request

e) there are k stale pages at the beginning, at the moment of the request there are $s(i) = k - i + 1$ stale pages not requested so far

f) $s(i) - c(i)$ of them are still in the fast memory

g) the expected cost is

$$\frac{s(i) - c(i)}{s(i)} \cdot 0 + \frac{c(i)}{s(i)} \cdot 1 \leq \frac{c}{s(i)}$$

h) by linearity of expectation the expected cost of the phase is at most

$$c + \sum_{i=1}^{s} \frac{c}{k - i + 1} \leq c + \sum_{j=2}^{k} \frac{c}{j} = c \cdot H_k$$

Yao's MINMAX Principle

the expected cost of a randomized algorithms for the worst input

$\geq$

the expected cost of the best deterministic algorithm for the input distribution q

$$\max_{x \in X} (E[c(A,x)]) \geq \min_{a \in A} (E(a,X))$$

Proof.
the best way is to build a rectangle where

i. each row corresponds to a deterministic algorithm

ii. each column corresponds to an input

iii. each entry corresponds to cost

iv. max weighted sum over over a column must be at least minimal weighted sum over a row

symbolic proof:
let $C = \max_{x \in X} (E[c(A,x)])$, $D = \min_{a \in A} (E(a,X))$

$$C = \sum_x q_x \cdot C \geq \sum_x q_x \cdot E[c(A,x)] = \sum_x q_x \cdot \sum_a p_a \cdot c(a,x)] =$$

$$\sum_a p_a \cdot \sum_x q_x \cdot c(a,x) \geq \sum_a p_a \cdot D = D$$

Version for online algorithms

the competitive ratio of the best randomized online algorithm against any oblivious adversary
$\geq$
the competitive ratio of the best deterministic online algorithm under a worst-case input distribution

`Theorem`
For any randomized online algorithm for paging has competitive ratio $\geq H_k$

`Proof`

i. use Yao's MinMax principle: it suffices to show competitive ratio for any deterministic algorithm for carefully chosen input distribution

ii. input distribution:
   - only $k+1$ pages used
   - for $t = 1$ choose page uniformly at random
   - for $t > 1$ choose a page uniformly at random from the set of all pages but not the last one chosen

iii. a phase defined so that within a phase requests to exactly $k+1$ distinct pages

iv. OPT has cost 1 per phase

v. deterministic algorithm:

   a) at each step the expected cost is $\frac{1}{k}$ as one of k pages that can be requested is not in the fast memory

   b) the expected length of a phase is $k \cdot H_k$

VI. RAPID MIXING ALGORITHMS

(mainly based on Randall's paper)

- **idea:** the algorithm is not computing something but performing a random walk in a certain space
- outcome:
  - final „random" position in the space
  - history of the walk indicating some value
  - sometimes a mixing based method of algorithm analysis
- typical scenario:
  - we wish to choose an element $s \in \mathcal{S}$ at random
  - the space $\mathcal{S}$ is complex, we have no way to enumerate its elements
  - ... but we know at least some (non-random) elements of $\mathcal{S}$
  - ... and can perform a random walk through $\mathcal{S}$ through small random modifications in the current state $s \in \mathcal{S}$
- challenges:
  i. are all $s \in \mathcal{S}$ reachable in this way?
  ii. what is the probability distribution of the position after t steps? Is it close e.g. to the uniform one over $\mathcal{S}$? How relevant is the starting position?
- examples: statistical physics: movements of a gas particle, shuffling of a deck of cards

Markov chain

- a (usually finite) set of states $\mathcal{S}$
- transition probabilities to change the state $P(i, j)$ is the probability to change the state from state s_i to s_j