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

- – check 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:

$\rightarrow$ **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")

$\rightarrow$ **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

$\rightarrow$ **concensus:** processes might have a different understanding of the global state

$\rightarrow$ **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    $S = \text{ParallelPrefix}(\{A[2i] + A[2i+1]\}_{i=1,...,|A|/2})$
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 $:= \{V[j] : \text{TABLE}(\text{hash}(V[j], m, i) = j)\}$
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 $\text{TABLE}[\text{hash}(k,m,i) \neq k]$
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)
  1 for $i = 0, ..., b-1$
  2 flags $:= \{(a \gg i) \bmod 2 : a \in A\}$
  3 notflags $:= 1 - \text{flags}$
  4 $R_0 := \text{parallelprefix}(\text{notflags})$
  5 $s_0 := \text{sum}(\text{notflags})$
  6 $R_1 := \text{paralleprefix}(\text{flags})$
  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: 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)
  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) ∈ E: child[u] = 1 and child[v] = 0}
5 put in LABELS values from HOOKS
6 $E' := \{(\text{LABELS}[u], \text{LABELS}[v]: (u, v) \in E \text{ and LABELS}[u] \neq \text{LABELS}[v]\}$
7 LABELS' := randomcontraction(LABELS, E')
8 insert to LABELS' entries (u, LABELS'[v]) for (u, v) ∈ HOOKS

LABELS' returned

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

II. Distributed computing

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