PARALLEL COMPUTING

Algorithms and Complexity

Armin Biere

2018/03/06

JOHANNES KEPLER
UNIVERSITY LINZ

Slow-Down in Parallel SAT

table 2 of

Parallel Multithreaded Satisfiability Solver: Design and Implementation.
Yulik Feldman, Nachum Dershowitz, Ziyad Hanna
`http://dx.doi.org/10.1016/j.entcs.2004.10.020`

- paper is inconclusive about the reason for slow-down
- probably more threads work on useless sub-tasks
- sharing clauses caching sub-computation increases pressure on memory system
- maybe search space splitting was not a good idea (guiding path)

Low Speedup in Parallel SAT

slide 4 of (video 3:30)

`http://www.birs.ca/events/2014/5-day-workshops/14w5101/videos/watch/201401221154-Sabharwal.html`

- sequential SAT algorithms produce proofs of large depth (= $span$)
- so need new algorithms which produce low depth proofs

Memory System is Good Enough

Martin Aigner, Armin Biere, Christoph Kirsch, Aina Niemetz, Mathias Preiner.
Analysis of Portfolio-Style Parallel SAT Solving on Current Multi-Core Architectures.
In Proc. Intl. Workshop on Pragmatics of SAT (POS'13),
EPiC Series in Computing, vol. 29, 28-40, EasyChair 2014.
`http://fmv.jku.at/papers/AignerBiereKirschNiemetzPreiner-POS13.pdf`

- largest speed-up obtained by portfolio approach
  - run different search strategies in parallel
  - if one terminates stop all
  - in practice share some important learned clauses caching sub-computations
- slow-down due to memory system?
  - since memory system (memory / caches / bus) are shared in multi-core systems
  - slow-down not too bad (particularly for solvers with small working set)
  - even though considered memory-bound (but random access)
  - waiting time for memory to arrive overlaps

Clever Splitting

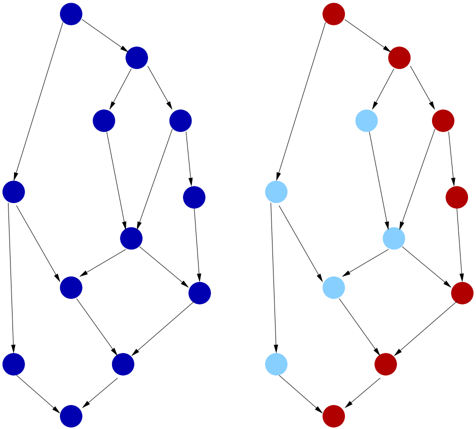
Marijn Heule, Oliver Kullmann, Siert Wieringa, Armin Biere.
Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads.
Haifa Verification Conference 2011: 50-65, Springer 2012
`http://dx.doi.org/10.1007/978-3-642-34188-5_8`

Marijn J.H. Heule, Oliver Kullmann, and Victor Marek
Solving and Verifying the boolean Pythagorean Triples problem via Cube-and-Conquer.
SAT 2016, 196-211, Springer 2016
`http://dx.doi.org/10.1007/978-3-319-40970-2_15`

Everything is Bigger in Texas
`https://www.cs.utexas.edu/~marijn/ptn/`
JKU CS Colloquium 22. June 2016

Work and Span

work
12



span
7

Amdahls Law with Work and Span

$T = work$ = sequential time $\qquad T_p$ = wall-clock time p CPUs $\qquad T_\infty$ = wall-clock time ∞ CPUs

Speedup $\quad S_P = T/T_P$

$span$ critical path $\quad$ (also called “makespan” in the context of scheduling)

f fraction of sequential work, thus $\quad f = span/work$

simplified Amdahl’s law in terms of $work$ and $span$: $\qquad S_p \leq 1/f = work/span$

Reduce $span$ as much as possible:

- ☐ keep sequential blocks short! $\Rightarrow$ fine grained locking is evil
- ☐ keep sequential dependencies short! $\Rightarrow$ (non-logarithmic) loops are evil

Pebble Games

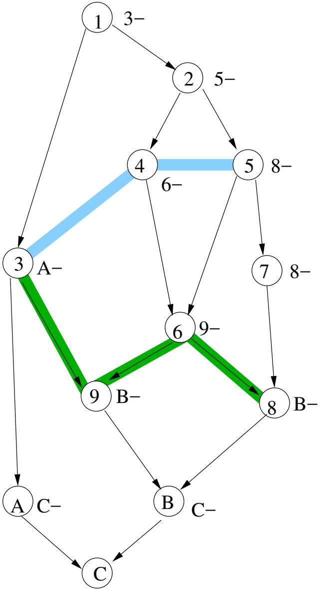
Given a directed acyclic graph with one sink.

Nodes of the graph have a pebble or not.

One **step** can either . . .
. . . remove a pebble from a node . . .
. . . or add a new pebble to a node without one, . . .
. . . but only if all its predecessor have a pebble.

Goal is to only have a pebble on the sink node.

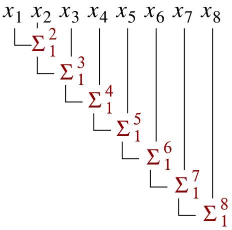
What is the smallest maximum number of pebbles needed?



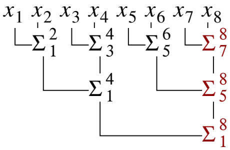
common concept in complexity theory
assuming intermediate results have to be stored
relates to smallest p needed to reach maximum speed-up
this version (black pebble game) actually only gives space bounds

Sum

compute sum $\sum_1^n x_i$ for n numbers x_i in parallel



- sequential
  - $y_0 = 0, \quad y_{i+1} = y_i + x_i$ for $i = 1 \ldots n-1$
  - $work = T = \mathcal{O}(n) \quad (n-1$ additions)
  - $span = \mathcal{O}(n)$ too
  - since y_{i+1} depends on all previous y_j with $j \leq i$
  - thus no speed-up $S_p = \mathcal{O}(1)$
- parallel
  - **associativity** allows to regroup computation
  - $work = \mathcal{O}(n)$ remains the same
  - $span = \mathcal{O}(\log n)$ reduces exponentially
  - speed-up not ideal but $S_n = \mathcal{O}(n/\log n)$
  - note $p > n$ does not make sense



Prefix / Scan

compute all sums $s_j = \sum_1^j x_i$ for all $j = 1 \ldots n$ and again n numbers x_i in parallel

sequential version as in previous slide

parallel version needs a second depth $\mathcal{O}(\log\ n)$ pass

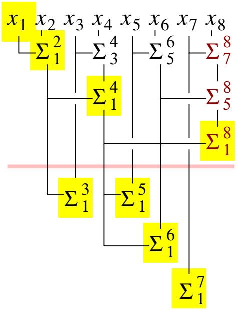
works even "in place" (first pass overwrites original x_i)

but actual "wiring" complicated

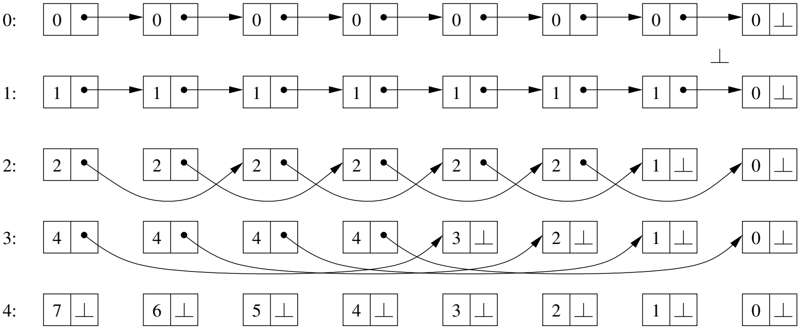
still $span = \mathcal{O}(\log\ n)$

basic algorithmic idea for many "parallel" algorithms

(propagate and generate adders with prefix trees instead of ripple carry adders)



List Ranking / Pointer Jumping



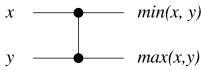
determine distance to head of list:

as long there is i with $next[i] \neq \bot$:
$val[i]$ += $val[next[i]]$
$next[i]$ = $next[next[i]]$

Sorting Networks

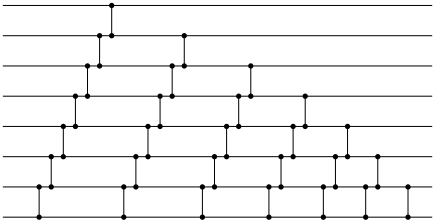
- circuits for sorting fixed number n of inputs
  - basic “gate” compare-and-swap:

    $cmpswap(x, y)$:= $(min(x, y), max(x, y))$



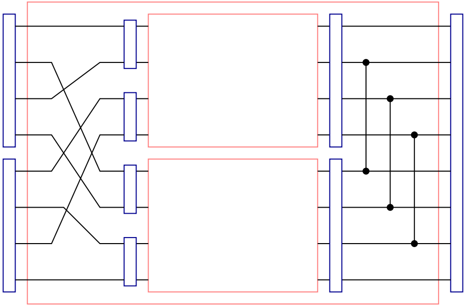
  - interesting challenge to get smallest sorting network
    for $n = 11$ size only known to be between 33 and 35 compare-and-swap operations
- zero-one principle
  - correctness of sorting network (it sorts!) ...
  - ... only requires sorting 0 and 1 inputs (bits) ...
  - ... as long only compare-and-swap is used.
- asymptotic complexity of algorithms
  - examples: Bitonic Sorting, Batcher Odd-Even Mergesort
  - with $span$ = $\mathcal{O}(\log^2 n)$
  - with $work$ = $\mathcal{O}(n \cdot \log^2 n)$ = T_1
  - but sequential time $T = \mathcal{O}(n \cdot \log n)$
  - maximum absolute speed-up $S_n = \mathcal{O}(n / \log n)$

Bubble Sort Example



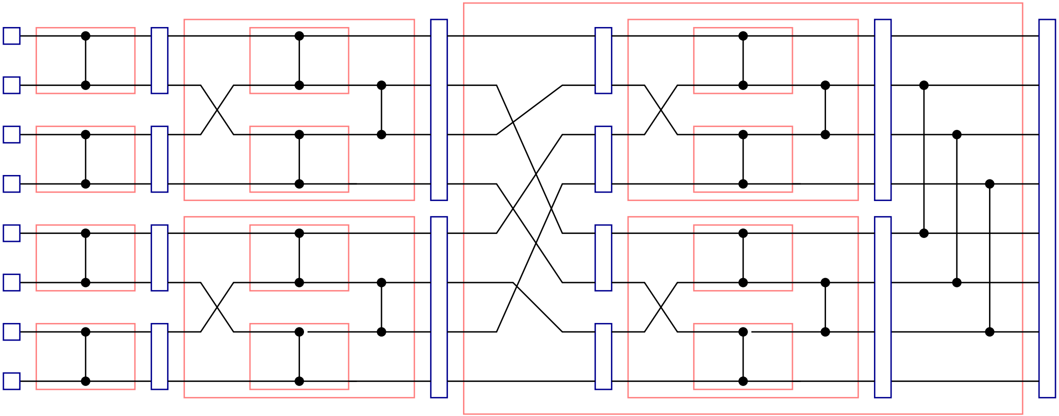
- top-most i sorted after i phases
- lowest value only sorted after $n-1$ compare-and-swaps
- $work = \mathcal{O}(n^2)$
- $span = \mathcal{O}(n)$
- looks like perfect speedup $S_n = \mathcal{O}(n)$ w.r.t. (bad) sequential algorithm
- however, if we compare against Quicksort $T = O(n \cdot \log n)$ we only get $S_n = \mathcal{O}(\frac{n \cdot \log n}{n}) = \mathcal{O}(\log n) < \mathcal{O}(n / \log n)$

Batcher Odd-Even Mergesort



- basically as mergesort
  - split input into two parts . . .
  - . . . sort parts recursively . . .
  - . . . merge sorted sequence.
- example: recursion for $n = 8$
  - outer block takes two sorted sequences of size 4 each
  - each inner block takes two sorted sequences of size 2 each
  - outer input sequences need to be sorted too

Batcher Odd-Even Mergesort



NC – Nick's Class

$f(n)$ polylogarithmic *iff* exists constant c such that $f(n) = \mathcal{O}(\log^c n)$

NC is set of decision problems ...
... which can be decided in polylogarithmic time ...
... on a parallel computer with polynomial many processors, e.g., ...
... exists constant c such that $p = \mathcal{O}(n^k)$.

NC^c requires (parallel) computation time ($span$) in $\mathcal{O}(\log^c n)$

NC = $\bigcup$ NC^c

L, NL, AC

L is set of decision problems solvable in logarithmic space determistically

NL is set of decision problems with logarithmic space non-determistically

NC = AC is the set of decision problems with logarithmic circuit complexity, i.e., . . .
. . . each input of size n can be decided by polynomial circuit with logarithmic depth in n, . . .
. . . made of gates with bounded (NC) or unbounded (AC) number of inputs

as before define NC^c and AC^c requiring $\mathcal{O}(\log^c n)$ depth (layers)

P Completeness

$$\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \subseteq \mathsf{AC}^2 \subseteq \mathsf{NC}^3 \subseteq \cdots \subseteq \mathsf{NC} = \mathsf{AC} \subseteq \mathsf{P}$$

using “logarithmic” reductions

it is commonly believed that $\mathsf{NC} \neq \mathsf{P}$

accordingly P-hard problems are supposed to be NOT “parallelizable”

similar to the common belief that $\mathsf{P} \neq \mathsf{NP}$

Circuit Evaluation Problem

Given a boolean circuit with one output, and an evaluation to its inputs.

Evaluate the circuit and determine its output value for that input assignment.

This problem (deciding whether output yields one) is P-complete . . .
. . . and thus considered **not** to be parallelizable.

Thus evaluating a function can **not** be done “effectively” in parallel.

One step of simulation or constraint propagation are **not** parallelizable! (?)