7

Parallel Algorithms

17 November 2023

Sebastian Wild

COMP526 (Fall 2023)
University of Liverpool

version `2023-11-16 22:18 H`

Learning Outcomes

1. Know and apply *parallelization strategies* for embarrassingly parallel problems.
2. Identify *limits of parallel speedups*.
3. Understand and use the *parallel random-access-machine* model in its different variants.
4. Be able to *analyze* and compare simple shared-memory parallel algorithms by determining *parallel time and work*.
5. Understand efficient parallel *prefix sum* algorithms.
6. Be able to devise high-level description of *parallel quicksort and mergesort* methods.

Unit 7: *Parallel Algorithms*

Outline

7 Parallel Algorithms

7.1 Parallel Computation

Types of parallel computation

£££ can't buy you more time . . . but more computers!

$\rightsquigarrow$ Challenge: Algorithms for *parallel* computation.

There are two main forms of parallelism:

1. **shared-memory parallel computer** ← *focus of today*
    - p *processing elements* (PEs, processors) working in parallel
    - **single** big memory, **accessible from every PE**
    - communication via shared memory
    - think: a big server, 128 CPU cores, terabyte of main memory
2. **distributed computing**
    - p PEs working in parallel
    - each PE has **private** memory
    - communication by sending **messages** via a network
    - think: a cluster of individual machines

PRAM – Parallel RAM

- extension of the RAM model (recall Unit 1)
- the p PEs are identified by ids $0, \ldots, p-1$
  - like w (the word size), p is a parameter of the model that can grow with n
  - $p = \Theta(n)$ is not unusual maaany processors!
- the PEs all **independently** run the same RAM-style program (they can use their id there)
- each PE has its own registers, but `MEM` is shared among all PEs
- computation runs in **synchronous** steps: in each time step, every PE executes one instruction
- As for RAM:
  - assume a basic "operating system"
  - ⇝ write algorithms in pseudocode instead of RAM assembly
  - **NEW:** loops and commands can be run **"in parallel"** (examples coming up)

PRAM – Conflict management

Problem: What if several PEs simultaneously overwrite a memory cell?

- **EREW**-PRAM (exclusive read, exclusive write)
  any **parallel access** to same memory cell is **forbidden** (crash if happens)
- **CREW**-PRAM (concurrent read, exclusive write)
  parallel **write** access to same memory cell is **forbidden**, *but reading is fine*
- **CRCW**-PRAM (concurrent read, concurrent write)
  concurrent access is allowed,
  need a rule for write conflicts:
  - common CRCW-PRAM:
    all concurrent writes to same cell must write **same** value
  - arbitrary CRCW-PRAM:
    some unspecified concurrent write wins
  - (more exist . . .)

- no single model is always adequate, but our default is CREW

PRAM – Execution costs

Cost metrics in PRAMs

- **space:** total amount of accessed memory
- **time:** number of steps till all PEs finish   assuming sufficiently many PEs!
  sometimes called *depth* or *span*
- **work:** total #instructions executed on **all** PEs

Holy grail of PRAM algorithms:

- minimal time (=span)
- work (asymptotically) no worse than running time of best sequential algorithm
  $\rightsquigarrow$ *"work-efficient"* algorithm: work in same Θ-class as best sequential

The number of processors

Hold on, my computer does not have $\Theta(n)$ *processors! Why should I care for span and work!?*

Theorem 7.1 (Brent's Theorem)

If an algorithm has span T and work W (for an arbitrarily large number of processors), it can be run on a PRAM with p PEs in time $O(T + \frac{W}{p})$ (and using $O(W)$ work). ◀

Proof: schedule parallel steps in round-robin fashion on the p PEs.

$\rightsquigarrow$ span and work give guideline for *any* number of processors

7.2 Parallel String Matching

Embarrassingly Parallel

- A problem is called *"embarrassingly parallel"*
  if it can immediately be split into *many, small subtasks*
  that can be solved completely *independently* of each other
- Typical example: sum of two large matrices (all entries independent)

⇝ best case for parallel computation (simply assign each processor one subtask)

- Sorting is not embarrassingly parallel
  - no obvious way to define many *small* (= efficiently solvable) subproblems
  - but: some subtasks of our algorithms are (stay tuned . . .)

Parallel string matching – Easy?

- We have seen a plethora of string matching methods in Unit 4
- But all efficient methods seem inherently sequential
  Indeed, they became efficient only after building on knowledge from previous steps!

Sounds like the *opposite* of parallel!

⇝ How well can we parallelize string matching?

Here: string matching = find *all* occurrences of P in T (more natural problem for parallel)
always assume $m \le n$

Subproblems in string matching:

- string matching = check all guesses $i = 0, \ldots, n - m - 1$
- checking one guess is a subtask!

Parallel string matching – Brute force

- Check all guesses in parallel

```
procedure parallelBruteForce($T[0..n)$, $P[0..m)$)
    for $i := 0, \ldots, n-m-1$ do in parallel ← only difference to normal brute force!
        for $j := 0, \ldots, m-1$ do
            if $T[i+j] \neq P[j]$ then break inner loop
        if $j == m$ then report match at $i$
    end parallel for
```

- PE k is executing the loop iteration where $i = k$.
  - ⇝ requires that all iterations can be done **independently**!
  - Different PEs work **in lockstep** (synchronized after each instruction)
  - similar to OpenMP `#pragma omp parallel for`
- checking whether *no* match was found by *any* PE more effort ⇝ . . . stay tuned

⇝ **Time**: $\Theta(m)$ using sequential checks
$\Theta(\log m)$ on CREW-PRAM (⇝ tutorials)
$\Theta(1)$ on CRCW-PRAM (⇝ tutorials)

Work: $\Theta((n-m)m)$ ⇝ not great
. . . much more than best sequential

Parallel string matching – Blocking

Divide T into **overlapping** blocks of $2m - 1$ characters:
$T[0..2m-1), T[m..3m-1), T[2m..4m-1), T[3m..5m-1) \ldots$

- Search all blocks in parallel, each using efficient *sequential* method

```
procedure blockingStringMatching($T[0..n)$, $P[0..m)$)
    for $b := 0, \ldots, \lceil n/m \rceil$ do in parallel
        $result$ := KMP($T[bm .. (b+1)m - 1)$, $P$)
        if $result \neq$ NO_MATCH then report match at $result$
    end parallel for
```

$\leadsto$ **Time**:

- loop body has text of length $n' = 2m - 1$ and pattern of length m

  $\leadsto$ KPM runtime $\Theta(n' + m) = \Theta(m)$

$\leadsto$ **Work**: $\Theta(\frac{n}{m} \cdot m) = \Theta(n)$ $\leadsto$ work efficient!

Parallel string matching – Discussion

- 👍 very simple methods
- 👍 could even run distributed with access to part of T

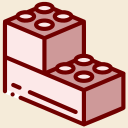
- 👎 parallel speedup only for $m \ll n$

- work-efficient methods with better parallel time possible?
  - ⇝ must genuinely parallelize the matching process! (and the preprocessing of the pattern)
  - ⇝ needs new ideas (much more complicated, but possible!)

- **Parallel string matching – State of the art:**
  - $O(\log m)$ time & work-efficient parallel string matching (very complicated)
    - this is optimal for CREW-PRAM
  - on CRCW-PRAM: matching part even in $O(1)$ time (easy)
    but preprocessing requires $\Theta(\log \log m)$ time (very complicated)

7.3 Parallel Primitives

Building blocks



- Most nontrivial problems need tricks to be parallelized
- Some versatile building blocks are known that help in many problems

⇝ We study some of them now, before we apply them to *parallel sorting*

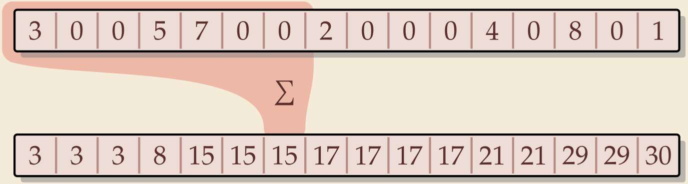
The following problems might not look natural at first sight . . . but turn out to be good abstractions.
⇝ *bear with me*

Prefix sums

Prefix-sum problem (also: cumulative sums, running totals)

- Given: array $A[0..n)$ of numbers
- Goal: compute all prefix sums $A[0] + \cdots + A[i]$ for $i = 0, \ldots, n-1$
  may be done "in-place", i. e., by overwriting A

Example:



input:	3	0	0	5	7	0	0	2	0	0	0	4	0	8	0	1
output:	3	3	3	8	15	15	15	17	17	17	17	21	21	29	29	30

Prefix sums – Sequential

- sequential solution does $n-1$ additions
- but: cannot parallelize them!
  ↯ data dependencies!

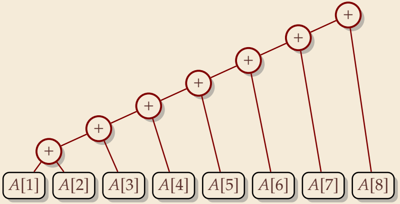
⇝ need a different approach

```
procedure prefixSum($A[0..n)$)
    for $i := 1, \dots, n-1$ do
        $A[i] := A[i-1] + A[i]$
```

Let's try a simpler problem first.

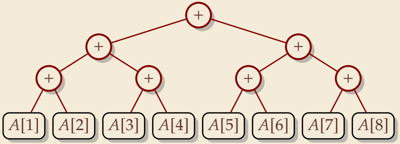
Excursion: Sum

- Given: array $A[0..n)$ of numbers
- Goal: compute $A[0] + A[1] + \cdots + A[n-1]$
  (solved by prefix sums)



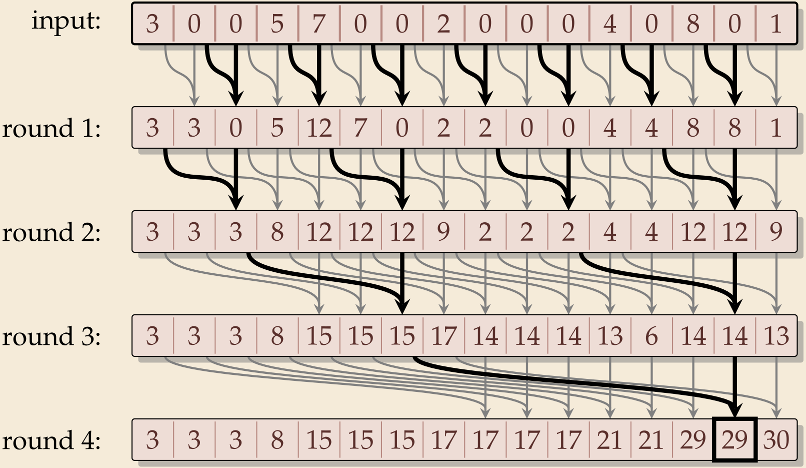
Any algorithm *must* do $n-1$ binary additions

⇝ Height of tree = parallel time!



Parallel prefix sums

- Idea: Compute all prefix sums with balanced trees in parallel
  Remember partial results for reuse



input:	3	0	0	5	7	0	0	2	0	0	0	4	0	8	0	1
round 1:	3	3	0	5	12	7	0	2	2	0	0	4	4	8	8	1
round 2:	3	3	3	8	12	12	12	9	2	2	2	4	4	12	12	9
round 3:	3	3	3	8	15	15	15	17	14	14	14	13	6	14	14	13
round 4:	3	3	3	8	15	15	15	17	17	17	17	21	21	29	29	30

Parallel prefix sums – Code

- can be realized in-place (overwriting A)
- assumption: in each parallel step, all reads precede all writes

```
procedure parallelPrefixSums($A[0..n)$)
    for $r := 1, \ldots \lceil \lg n \rceil$ do
        $step := 2^{r-1}$
        for $i := step, \ldots n-1$ do in parallel
            $x := A[i] + A[i - step]$
            $A[i] := x$
        end parallel for
    end for
```

Parallel prefix sums – Analysis

- **Time:**
  - all additions of one round run in parallel
  - $\lceil \lg n \rceil$ rounds
  - ⇝ $\Theta(\log n)$ time best possible!
- **Work:**
  - $\geq \frac{n}{2}$ additions in all rounds (except maybe last round)
  - ⇝ $\Theta(n \log n)$ work
  - more than the $\Theta(n)$ sequential algorithm!
- Typical trade-off: greater parallelism at the expense of more overall work
- For prefix sums:
  - can actually get $\Theta(n)$ work in *twice* that time!
  - ⇝ algorithm is slightly more complicated
  - instead here: linear work in *thrice* the time using "blocking trick"

Work-efficient parallel prefix sums

↙ recall string matching!

standard trick to improve work: compute small blocks sequentially

1. Set $b := \lceil \lg n \rceil$
2. For blocks of b consecutive indices, i. e., $A[0..b)$, $A[b..2b)$, … **do in parallel**:
   - compute local prefix sums with fast **sequential** algorithm
3. Use previous work-inefficient parallel algorithm only on **rightmost elements** of block, i. e., to compute prefix sums of $A[b-1]$, $A[2b-1]$, $A[3b-1]$, …
4. For blocks $A[0..b)$, $A[b..2b)$, … do in parallel:
   Add block-prefix sums to local prefix sums

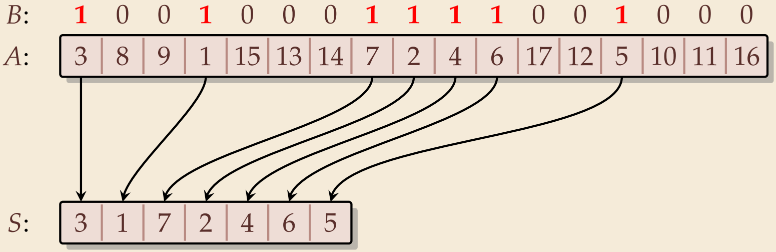
Analysis:

- **Time:**
  - 2. & 4.: $\Theta(b) = \Theta(\log n)$ time
  - 3. $\Theta(\log(n/b)) = \Theta(\log n)$ time
- **Work:**
  - 2. & 4.: $\Theta(b)$ per block $\times$ $\lceil \frac{n}{b} \rceil$ blocks $\rightsquigarrow$ $\Theta(n)$
  - 3. $\Theta\left(\frac{n}{b}\log\left(\frac{n}{b}\right)\right) = \Theta(n)$

Compacting subsequences

How do prefix sums help with sorting?   one more step to go . . .

Goal: *Compact* a subsequence of an array



B: **1** 0 0 **1** 0 0 0 **1** **1** **1** **1** 0 0 **1** 0 0 0

A: 3 | 8 | 9 | 1 | 15 | 13 | 14 | 7 | 2 | 4 | 6 | 17 | 12 | 5 | 10 | 11 | 16

S: 3 | 1 | 7 | 2 | 4 | 6 | 5

Use prefix sums on bitvector B

⇝ offset of selected cells in S

```
C := B // deep copy of B
parallelPrefixSums(C)
for $j := 0, \ldots, n-1$ do in parallel
    if $B[j] == 1$ then $S[C[j]-1] := A[j]$
end parallel for
```

7.4 Parallel Sorting

Parallel Mergesort

- Recursive calls can run in parallel (data independent)!
- how about merging sorted halves $A[l..m)$ and $A[m..r)$?
- Our pointer-based sequential method seems hard to parallelize

$\leadsto$ Must treat all elements independently.

#elements $\le x$

  - correct position of x in sorted output = *rank* of x breaking ties by position in A
  - # elements $\le x$ = # elements from $A[l..m)$ that are $\le x$
    \+ # elements from $A[m..r)$ that are $\le x$
  - rank in **own run** is simply the **index** of x in that run!
  - find rank in **other** run by *binary search*

  $\leadsto$ can move x directly to correct position

Parallel Mergesort – Code

```
procedure parMergesort($A[l..r)$, $buf$)
    $m := l + \lfloor (r - l)/2 \rfloor$
    in parallel { parMergesort($A[l..m)$, $buf$), parMergesort($A[m..r)$, $buf$) }
    parallelMerge($A[l..m)$, $A[m..r)$, $buf$)
    for $i = l, \dots, r-1$ do in parallel // copy back in parallel
        $A[i] := buf[i]$
    end parallel for

procedure parallelMerge($A[l..m)$, $A[m..r)$, $buf$)
    for $i = l, \dots, m-1$ do in parallel
        $r := (i - l)$ + binarySearch($A[m..r)$, $A[i]$)  // binarySearch($A$, $x$) returns #elements $< x$ in $A$
        $buf[r] = A[i]$
    end parallel for
    for $j = m, \dots, r-1$ do in parallel
        $r :=$ binarySearch($A[l..m)$, $A[j]$) $+ (j - m)$
        $buf[r] = A[j]$
    end parallel for
```

Parallel mergesort – Analysis

- **Time:**
  - merge: $\Theta(\log n)$ from binary search, rest $O(1)$
  - mergesort: depth of recursion tree is $\Theta(\log n)$

  $\rightsquigarrow$ total time $O(\log^2(n))$
- **Work:**
  - merge: n binary searches $\rightsquigarrow$ $\Theta(n \log n)$

  $\rightsquigarrow$ mergesort: $O(n \log^2(n))$ work

  - work can be reduced to $\Theta(n)$ for merge (complicated!)
    - do full binary searches only for regularly sampled elements
    - ranks of remaining elements are sandwiched between sampled ranks
    - use a sequential method for small blocks, treat blocks in parallel
    - (details omitted)

Parallel Quicksort

Let's try to parallelize Quicksort

- As for Mergesort, recursive calls can run in parallel ✓
- our sequential partitioning algorithm seems hard to parallelize
- but can split partitioning into *phases*:
  1. **comparisons:** compare all elements to pivot (in parallel), store result in bitvectors
  2. compute prefix sums of bit vectors (in parallel as above)
  3. **compact** subsequences of small and large elements (in parallel as above)

Parallel Quicksort – Code

```
procedure parQuicksort($A[l..r]$)
    $b$ := choosePivot($A[l..r]$)
    $j$ := parallelPartition($A[l..r]$, $b$)
    in parallel { parQuicksort($A[l..j]$), parQuicksort($A[j+1..r]$) }

procedure parallelPartition($A[0..n)$, $b$)
    swap($A[n-1]$, $A[b]$);  $p$ := $A[n-1]$
    for $i = 0, \ldots, n-2$ do in parallel
        $S[i]$ := $\big[A[i] \le p\big]$    // $S[i]$ is 1 or 0
        $L[i]$ := $1 - S[i]$
    end parallel for
    in parallel { parallelPrefixSum($S[0..n-2]$);  parallelPrefixSum($L[0..n-2]$) }
    $j$ := $S[n-2] + 1$
    for $i = 0, \ldots, n-2$ do in parallel
        $x$ := $A[i]$
        if $x \le p$ then $A[S[i]-1]$ := $x$
        else $A[j + L[i]]$ := $x$
    end parallel for
    $A[j]$ := $p$
    return $j$
```

Parallel Quicksort – Analysis

- **Time:**
  - partition: all $O(1)$ time except prefix sums $\rightsquigarrow$ $\Theta(\log n)$ time
  - Quicksort: expected depth of recursion tree is $\Theta(\log n)$

  $\rightsquigarrow$ total time $O(\log^2(n))$ in expectation
- **Work:**
  - partition: $O(n)$ time except prefix sums $\rightsquigarrow$ $\Theta(n)$ work (with work-efficient prefix-sums algorithm)

  $\rightsquigarrow$ Quicksort $O(n \log(n))$ work in expectation
  - (expected) work-efficient parallel sorting!

Parallel sorting – State of the art

- more sophisticated methods can sort in $O(\log n)$ parallel time on CREW-PRAM
- practical challenge: small units of work add overhead
- need a lot of PEs to see improvement from $O(\log n)$ parallel time

$\rightsquigarrow$ implementations tend to use simpler methods above

  - check the Java library sources for interesting examples!
    `java.util.Arrays.parallelSort(int[])`