1 11 тчг ALGORITHMS \$ EFFIC F. CIENTALGORITHMS \$ EFF CIENTAL G S FF R. HMS Ω ENTALGORITHMS\$E F F FICIENTALGO R. I E F тнм S R ਜ E Δ Т Ν G S G

# **Parallel Algorithms**

16 December 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25) Philipps-Universität Marburg version 2024-11-15 14:02 H

#### **Learning Outcomes**

#### Unit 10: Parallel Algorithms

- 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.

#### Outline

# **10** Parallel Algorithms

- **10.1** Parallel Computation
- 10.2 Parallel String Matching
- **10.3 Parallel Primitives**
- **10.4** Parallel Sorting

**10.1 Parallel Computation** 

### Types of parallel computation

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

→ 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 2)

- 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"
  - $\rightsquigarrow\,$  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 sometimes called *depth* or *span*

assuming sufficiently many PEs!

• 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
   *"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 10.1 (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.

→ span and work give guideline for *any* number of processors

# **10.2 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)
- $\rightsquigarrow best \ case \ for \ parallel \ computation \qquad (simply \ assign \ each \ processor \ one \ subtask)$
- 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 6
- 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, ..., n m 1
- checking one guess is a subtask!

### Parallel string matching – Brute force

Check all guesses in parallel

```
1procedure parallelBruteForce(T[0..n), P[0..m))2for i := 0, ..., n - m - 1 do in parallel <br/>sonly difference to normal brute force!3for j := 0, ..., m - 1 do4if T[i + j] \neq P[j] then break inner loop5if j == m then report match at i6end parallel for
```

• PE k is executing the loop iteration where i = k.

- $\rightsquigarrow~$  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

**Work:**  $\Theta((n - m)m) \rightsquigarrow$  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)...

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

**procedure** blockingStringMatching(T[0..n), P[0..m)) **for**  $b := 0, ..., \lceil n/m \rceil$  **do** in parallel **result** := KMP(T[bm .. (b+1)m - 1), P)

- <sup>4</sup> **if** *result* ≠ N0\_MATCH **then** report match at *result*
- 5 end parallel for

 $\rightsquigarrow$  Time:

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

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

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

### Parallel string matching – Discussion

very simple methods

 $\square$  could even run distributed with access to part of *T* 

 $\bigcirc$  parallel speedup only for  $m \ll n$ 

work-efficient methods with better parallel time possible?

- $\rightsquigarrow must genuinely parallelize the matching process! \qquad (and the preprocessing of the pattern)$
- → 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 Θ(log log m) time (very complicated)

## **10.3 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.  $\rightsquigarrow$  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] + · · · + A[i] for i = 0, . . . , n − 1 may be done "in-place", i. e., by overwriting A

#### Example:



### **Prefix sums – Sequential**

- sequential solution does n 1 additions
- but: cannot parallelize them!data dependencies!
- $\rightsquigarrow$  need a different approach
- Let's try a simpler problem first.

#### Excursion: Sum

- ► Given: array *A*[0..*n*) of numbers
- ► Goal: compute A[0] + A[1] + · · · + A[n 1] (solved by prefix sums)

Any algorithm *must* do n - 1 binary additions

 $\rightsquigarrow$  Height of tree = parallel time!

procedure prefixSum(A[0..n))

for 
$$i := 1, ..., n - 1$$
 do

3

A[i] := A[i-1] + A[i]



### Parallel prefix sums

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



### Parallel prefix sums – Code

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

```
1procedure parallelPrefixSums(A[0..n))2for r := 1, ... \lceil \lg n \rceil do3step := 2^{r-1}4for i := step, ... n - 1 do in parallel5x := A[i] + A[i - step]6A[i] := x7end parallel for8end for
```

### Parallel prefix sums – Analysis

#### Time:

- all additions of one round run in parallel
- ▶ [lg n] rounds
- $\rightsquigarrow \Theta(\log n)$  time best possible!

#### Work:

- ▶  $\geq \frac{n}{2}$  additions in all rounds (except maybe last round)
- $\rightsquigarrow \Theta(n \log n)$  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!
- $\rightsquigarrow~$  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*[2*b* 1], *A*[3*b* 1], . . .
- **4.** For blocks *A*[0..*b*), *A*[*b*..2*b*), . . . 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.  $\Theta\left(\frac{n}{b}\log(\frac{n}{b})\right) = \Theta(n)$

#### **Compacting subsequences**

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

Goal: Compact a subsequence of an array



Use prefix sums on bitvector B

 $\rightsquigarrow$  offset of selected cells in S

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

## **10.4 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
- $\rightsquigarrow~$  Must treat all elements independently.
  - correct position of x in sorted output = rank of x breaking ties by position in A

,#elements  $\leq x$ 

- ▶ # elements  $\leq x$  = # elements from A[l..m) that are  $\leq x$ + # elements from A[m..r) that are  $\leq x$
- rank in own run is simply the index of x in that run!
- ▶ find rank in **other** run by *binary search*
- $\rightsquigarrow$  can move *x* directly to correct position

### Parallel Mergesort – Code

```
procedure parMergesort(A[l..r), buf)
       m := l + |(r - l)/2|
2
       in parallel { parMergesort(A[l..m), buf), parMergesort(A[m..r), buf) }
3
       parallelMerge(A[1..m), A[m..r), buf)
 4
       for i = l, ..., r - 1 do in parallel // copy back in parallel
5
           A[i] := buf[i]
 6
       end parallel for
7
8
9 procedure parallelMerge(A[1..m), A[m..r), buf)
       for i = l, \ldots, m - 1 do in parallel
10
            r := (i - l) + \text{binarySearch}(A[m..r), A[i]) // binarySearch(A, x) returns #elements < x in A
11
           buf[r] = A[i]
12
       end parallel for
13
       for j = m, \ldots, r-1 do in parallel
14
            r := \text{binarySearch}(A[l..m), A[j]) + (j - m)
15
            buf[r] = A[i]
16
       end parallel for
17
```

### 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

1 **procedure** parQuicksort(A[l..r))

```
b := choosePivot(A[l..r))
2
      i := parallelPartition(A[l..r), b)
3
       in parallel { parQuicksort(A[1..i)), parQuicksort(A[i + 1..r)) }
 4
5
6 procedure parallelPartition(A[0..n), b)
       swap(A[n-1], A[b]); p := A[n-1]
7
       for i = 0, \ldots, n - 2 do in parallel
8
           S[i] := [A[i] \le p] // S[i] \text{ is 1 or 0}
9
           L[i] := 1 - S[i]
10
       end parallel for
11
       in parallel { parallelPrefixSum(S[0..n-2]); parallelPrefixSum(L[0..n-2]) }
12
      i := S[n-2] + 1
13
       for i = 0, \ldots, n - 2 do in parallel
14
           x := A[i]
15
           if x \leq p then A[S[i] - 1] := x
16
           else A[i + L[i]] := x
17
       end parallel for
18
       A[j] := p
19
       return j
20
```

### **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
- → implementations tend to use simpler methods above
  - check the Java library sources for interesting examples! java.util.Arrays.parallelSort(int[])