10 Parallel Algorithms

12 January 2026

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg
version 2026-01-12 16:04

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

Clicker Question

Have you ever written a concurrent program (explicit threads, job pools library, or using a framework for distributed computing)?

- **A** Yes
- **B** No
- **C** Concur. . . what?

→ *sli.do/cs566*

Types of parallel computation

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

⇝ Challenge: Algorithms for *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 ***p**rocessing 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

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"
  - ⇝ 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

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

Clicker Question

Does every computational problem allow a work-efficient algorithm?

A Yes

B No

→ *sli.do/cs566*

Clicker Question

Does every computational problem allow a work-efficient algorithm?

- **A** Yes ✓
- **B** ~~No~~

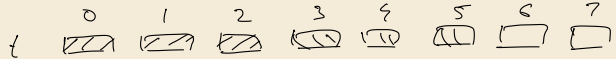
→ *sli.do/cs566*

The number of processors

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

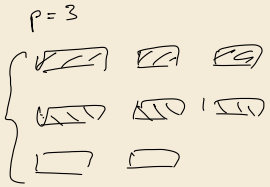
Theorem 10.1 (Brent's Theorem)
If an algorithm has span T and work W (for an arbitrarily large number of processors), it can be run on a PRAM with p PEs in time $O(T + \frac{W}{p})$ (and using $O(W)$ work). ◀

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



$\hat{p}$ = # PEs in PRAM

$T_p = \left\lceil \frac{\hat{p}}{p} \right\rceil$ steps



#rounds $= \Theta(T + \frac{W}{p})$

$$T \cdot \left\lceil \frac{W}{T_p} \right\rceil \leq T\left(\frac{W}{T_p} + 1\right) = \frac{W}{p} + T$$

⇝ 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)

⇝ 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 . . .)

Clicker Question

Is the string-matching problem "embarrassingly parallel"?

- **A** Yes
- **B** No
- **C** Only for $n \gg m$
- **D** Only for $n \approx m$

→ *sli.do/cs566*

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 $m \leq n$

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 $m \leq n$

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

```
1 procedure parallelBruteForce(T[0..n), P[0..m)):
2     for i := 0, ..., n − m − 1 do in parallel   ← only difference to normal brute force!
3         for j := 0, ..., m − 1 do
4             if T[i + j] ≠ P[j] then break inner loop
5         if j == m then report match at i
6     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

Parallel string matching – Brute force

- Check all guesses in parallel

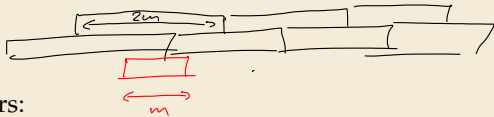
```
1 procedure parallelBruteForce(T[0..n), P[0..m)):
2     for i := 0, ..., n − m − 1 do in parallel   ← only difference to normal brute force!
3         for j := 0, ..., m − 1 do
4             if T[i + j] ≠ P[j] then break inner loop
5         if j == m then report match at i
6     end parallel for
```

- 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

```
1 procedure blockingStringMatching(T[0..n), P[0..m)):
2     for b := 0, ..., ⌈n/m⌉ do in parallel
3         result := KMP(T[bm .. (b+1)m − 1), P)
4         if result ≠ NO_MATCH then report match at result
5     end parallel for
```

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

```
1 procedure blockingStringMatching(T[0..n), P[0..m)):
2     for b := 0, ..., ⌈n/m⌉ do in parallel
3         result := KMP(T[bm .. (b+1)m − 1), P)
4         if result ≠ NO_MATCH then report match at result
5     end parallel for
```

$\Theta(m)$

$\rightsquigarrow$ **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!

Clicker Question

Is the string-matching problem "embarrassingly parallel"?

- **A** Yes
- **B** No
- **C** Only for $n \gg m$
- **D** Only for $n \approx m$

→ *sli.do/cs566*

Clicker Question

Is the string-matching problem "embarrassingly parallel"?

- **A** ~~Yes~~
- **B** ~~No~~
- **C** Only for $n \gg m$ ✓
- **D** ~~Only for $n \sim m$~~

→ *sli.do/cs566*

Parallel string matching – Discussion

- very simple methods
- could even run distributed with access to part of T
- parallel speedup only for $m \ll n$

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 – 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:** ← exam
  - $O(\log m)$ 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)