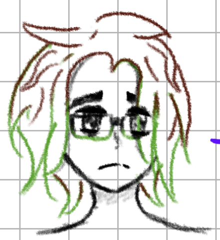
"ENC-Die Fortsetzung"

Disclaimer

Das ist nur eine Zusammenfassung und kein Ersatz für das Skriptum / die VOs. Keine Garantie darauf, dass alles so stimmt, wie es hier steht. Das ist nur meine Interpretation der Inhalte (meines Verständnisses nach). Falls etwas unklar sein sollte, bitte im Skriptum nachschauen. Third Block ist nicht inkludiert, da dieser nicht Teil vom Prüfungsstoff ist.



Viel ~~Spaß~~ ERFOLG beim ~~Lernen~~ BETEN!

(und viel Glück beim Test)

Inhaltsverzeichnis

1. Block: Motivation, Grundlagen, Modelle (PRAM)

"Free Lunch" phenomenon = Moore's Law: exponential increase (doubles every 18 months) of sequential performance (1970 - 2000)

FLOPS = FLoating point OPerations per Second: measure of nominal processor performance (multiplied by the number of cores)

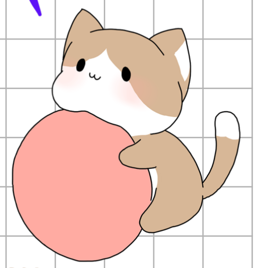
performance of single processor core: FLOP per clock cycle (determined by architecture) & clock frequency (number of "ticks" per second, measured in GHz)

whether it is reached depends on: 1. mixture of the operations and dependencies
2. ability of memory system to keep processor busy

Current terminology: CPU - processor / Central Processing Unit
consists of one/multiple/many
PE/PU - Processing Element / Unit / processor-core

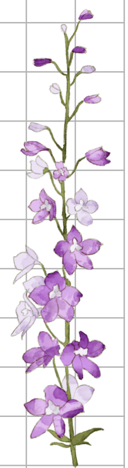
- multi-core CPU: few (2-32) cores
- many-core CPU: many cores, e.g. GPU — graphics processing unit (Graphikkarte)

Parallel Computing:
- focused on problem solving efficiency
- assumes dedication of whole computer system, without failure



Distributed Computing:
- focused on availability of resources
- resources may be spatially distributed, change or fail
- no centralized controll

Concurrent Computing:
- focused on concurrency and establishing correctness
- no centralized controll

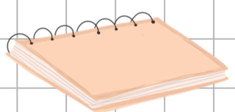


Bridging Model: suitable model for predictive, implementable results
↳ minimum requirement for good ones: better algorithm performance corresponds to better implementation performance
↳ performance portability: preservation of performance regardless of the system
↳ problematic for parellel computing, unproblematic for sequential computing

PRAM: Parallel Random Access Machine
↳ uses fixed number of processors which execute in lock-step (synchronized, one instruction per time step)
↳ machine is always in well-defined state
↳ purely theoretical, no realisation has been entirely successful so far

Variations of concurrent Access (for parellel memory access):
↳ when violated, maschine must stop executing program

1. EREW – Exclusive Read, Exclusive Write (concurrent = stimultaneous)
2. CREW – Concurrent Read, Exclusive Write
3. CRCW – Concurrent Read, Concurrent Write
   a) Common: all CPUs write the same value
   b) Arbitrary: either value survives
   c) Priority: the value of the highest priority CPU survives



Theorems:
1. On Common CRCW PRAM with n^2 processors (performing $O(n^2)$ operations), $O(1)$ parellel time steps suffice to find the maximum of n numbers stored in an array.
   ↳ n^2 element pair comparisons in 1 parellel step
   ↳ knock out non-maximum elements with outcome
2. On CREW PRAM with $\frac{n}{2}$ processors (performing $O(n)$ operations), $O(\log n)$ parellel time steps suffice to find the maximum of n numbers stored in an array
   ↳ $\lceil \log_2 n \rceil$ iterations, compares $\lfloor \frac{n}{2} \rfloor$ pairs per iteration, $\lceil \log_2 n \rceil$ iterations
3. On CREW PRAM, $O(lnm)$ operations and $O(l)$ parellel time steps suffice to multiply two $n \times l$ and $l \times m$ matrices into an $n \times m$ matrix
   ↳ can run on EREW PRAM with extra space

Flynn's Taxonomy: characterization of parallel maschines/models

↳ SISD – Single-Instruction, Single-Data: sequential computers

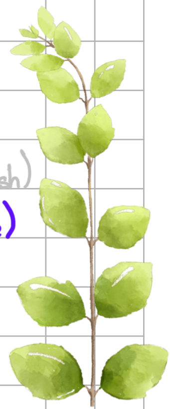
↳ SIMD – Single-Instruction, Multiple-Data: vector computers (works on short vectors of a few words)

↳ MISD – Multiple-Instruction, Single-Data: deeply pipelined system (single data stream passes stages)

↳ MIMD – Multiple-Instruction, Multiple-Data: PRAM maschines (each CPU can execute own instruction(s))

↳ SPMD – Single-Program, Multiple-Data: all CPUs execute the same program

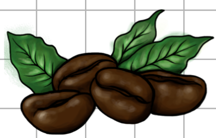
2. Block: Leistung paralleler Systeme und Algorithmen



Sequential vs. Parallel Time: Seq, Par................Algorithms (sequential, parallel)

$T_{seq}(n)$, $T_{par}(n,p)$........Running Times (time for the last core to finish)

n, p..........................input size (fixed), amount of cores (variable)



↳ difficult to measure without clear description of set up

↳ claims and observations must be objectively verified

Absolute Speed-Up: $S_p(n) = \frac{T_{seq}(n)}{T_{par}(n,p)}$ – speed-up of Par over Seq for input size $O(n)$

– speed-up of $O(p)$ = linear = perfect (hardly achievable)

– super-linear speed-up sometimes reported

– therefore it holds: $T_{par}(p,n) \geq \frac{T_{seq}(n)}{p}$

Cost: $pT_{par}(p,n)$ **Work:** $W_{par}(p,n)$ – total number of operations carried out for all p cores

(has linear speed-up per definition, ONLY cost-efficient algorithms have such

Cost-optimal parallel algorithm: $pT_{par}(p,n)$ is $O(T_{seq}(n))$ for best-known Seq

(CAN have linear speed-up, if cost-efficient

Work-optimal parallel algorithm: $W_{par}(p,n)$ is $O(T_{seq}(n))$ for best-known Seq

Relative Speed-Up: $SRel_p(n) = \frac{T_{par}(n,1)}{T_{par}(n,p)}$ – ratio of running time on one core to running time on p-cores

↳ fastest running time Par can achieve: $T\infty(n)$

↳ per Def. $T_{par}(n,p) \geq T\infty(n)$, therefore $SRel_p(n) = \frac{T_{par}(n,1)}{T_{par}(n,p)} \leq \frac{T_{par}(n,1)}{T\infty(n)}$ ← parallelism = largest speed-up & largest p where relative, linear speed-up is possible

↳ per Def. $T_{seq}(n) \leq T_{par}(1,n)$, therefore $S_p(n) \leq SRel_p(n)$

Overhead: work caused by parallelization, which isn't necessary in Seq ⇒ usally increase with bigger p

↳ if larger then $W_{seq}(n)$, Par can't have linear speed-up

Granularity: intervals between communication/synchronization operations

↳ coarse grained: rare com/synch; big intervals

↳ fine grained: frequent com/synch; small intervals

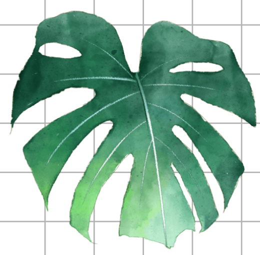
Load imbalance $= \max_{0\le i,j<p} |T_{par}(i,n) - T_{par}(j,n)|$ for $T_{par}(i,n)$ = running time of some processor i

↳ good balance: $T_{par}(i,n) \approx T_{par}(j,n)$ for all processors i,j

↳ static Load-balancing: work can be divided between CPUs

- ↳ oblivious slb: division based on input size and structure
  - ↳ embarassingly, trivially or pleasantly parallel
- ↳ adaptive slb: input and preprocessing needed for division

↳ dynamic Load-balancing: CPUs have to communicate & exchange work



Ahmdal's Law: Assuming that $W_{seq}(n)$ can be divided in strictly sequential fraction s (0<s≤1, independent of n) and parallelizable fraction $r=(1-s)$, the maximum speed-up (Par over Seq) is then $\frac{1}{s}$

↳ Seq's which fall under Amdahl's law have therefore limited speed-up ⇒ can't be used as Par

↳ Analysis Tool: if large s, new Seq is needed where s decreases with n

↳ Typical victims: IO-operations, seq. preprocessing, maintaining seq. data structures, long dependency chains

Running time of algorithm with linear speed-up: $T_{par}(p,n) = O(\underbrace{T(n)}_{\text{parallizable}}/p + \underbrace{t(n)}_{\text{non-parallizable}})$, for $t(n) = T\infty(n)$

↳ without linear speed-up: $T_{par}(p,n) = O(T(n)/f(p) + t(n))$, for $f(p) < p$ ← f(p) in o(p)

Scaled speed-up (against $T_{seq}(n) = O(T(n))$): $S_p(n) = \frac{T_{seq}(n)}{T_{par}(s,n)} = O\left(\frac{T(n)}{T(n)/p + t(n)}\right) = O\left(\frac{1}{1/p + t(n)/T(n)}\right) \to O(p)$

Performance of work-optimal algorithm: $T_{par}(p,n) = O(T(n)/p + t(n,p))$ for $t(n,p), T_{seq}(n)$ in $O(T(n))$ (for fixed p)

Parallel Efficiency: $E_p(n) = \frac{T_{seq}(n)}{p\,T_{par}(p,n)} = \frac{S_p(n)}{p}$, with $E_p(n) \le 1$, $S_p(n) = O(p)$ (linear speed-up) for $E_p(n) = e$ (some constant e), cost-opt. Alg. have const. E_p (opt = optimal, const = constant, Alg. = Algorithms)

Weak Scalability (relative to Seq):
- for constant e $\exists f(p)$ such that $E_p(n) = e$ for n in $\Omega(f(p))$, with $f(p)$ = iso-efficiency
- by keeping $\underbrace{T_{seq}(n)/p}_{\text{average work per CPU}} = w$, $T_{par}(p,n)$ remains constant. Input size: $g(p) = T_{seq}^{-1}(pw)$

Scaling Analysis:
- strong: Keep n constant. Alg. is strongly scalable if T_{par} is decreasing proportionally with p
- weak: Keep w constant by increasing n. Alg. is weakly scalable if T_{par} remains constant

4. Block: Parallele Algorithmen, Beispiele

merging problem: strictly sequential implementation

↳ complexity: $\Theta(n+m) = T_{seq}(n)$

```
void seq_merge(double A[], int n, double B[], int m, double C[]) {
  int i, j, k;

  i = 0; j = 0; k = 0;

  while (i<n&&j<m) {
    C[k++] = (A[i] <= B[j]) ? A[i++] : B[j++];
  }

  while (i<n) C[k++] = A[i++];
  while (j<m) C[k++] = B[j++];
}
```

A.length, B.length, resulting array with n+m elements

stable

stable merging/sorting algorithms: preservation of relative order of equal elem.

(assures stability but costs extra space & work)

assuming distinctness: merge triplets (x, F, i)

$x = A[i]$ or $B[i]$, F = flag whether $x = A[i]$ or $x = B[i]$

use lexicographic order $(x,F,i) < (x',F',i')$ if

$(x \le x' \vee (F = F' \wedge F = A))$ or $(x = x' \vee F = F' \vee i < i')$

avoid if possible, stability should be available by design!

merging by ranking: 1. foreach $(a : A)$ {
find $\text{rank}(A[i], B) = \text{position } j$ with $B[j] < A[i] < B[j+1]$
}

("rank of $A[i]$ in B" — computed with binary search in $O(\log m)$)

(indicates position of $A[i]$ in C: $i + \text{rank}(A[i], B)$)

(short: mbr) — classical load balancing problem

2. same for each $B[i]$ in B

→ sequential complexity: $O(n \cdot \log(m) + m \cdot \log(n)) = O((n+m)\log(\max(n,m)))$

parallel version (mbr): 1. assign each element a processor
2. let every processor compute the rank

(CREW or CRCW necessary)

→ parallel complexity: $O(\log(\max(n,m)))$ with $n+m$ processors

→ not work-optimal: work remains in $O((n+m)\log(\max(n,m)))$

improved parallel mbr: 1. Divide array into disjoint, consecutive blocks of size $\frac{n}{p}$
2. Compute $\text{rank}(A[b_x], B) = r$, for b_x = index of first element of x-th block
3. Merge A-block with B, determined by r and $A[b_x]$ by using sequential mbr

→ worsened complexity: $O(p \cdot \log(\max(n,m))) + O(n+m)$

(p processors aiming to rank $O(p)$ elements (load balancing problem))

solutions to this problem: 1. Bad Segment: big intervall between $\text{rank}(A[b_x], B)$ and $\text{rank}(A[b_{x+1}], B)$
Divide bad segment into p blocks of size $\frac{m}{p}$ in B
Compute ranks: $\text{rank}(B[b_x], A)$ for $0 \le x \le p$ parallely
↳ All ranks lie within the A-block(s) responsible for the bad segment
↳ Said A-blocks all have size at most $\frac{n}{p} + \frac{m}{p} = \frac{n+m}{p}$ ⇒ seq. mbr in $O(\frac{n+m}{p})$

(3. solution: prefix-sums ⇒ see p. 6)

2. Divide both A and B in blocks of size $\frac{n}{p}$ and $\frac{m}{p}$
Compute ranks: $\text{rank}(B[b_x], A)$ and $\text{rank}(A[b_x], B)$, for $0 \le x \le p$
Merge 2p pairs of blocks of size at most $\frac{n+m}{p}$ sequentially or parallely

Theorem: A and B can be merged work-optimally in $O(\frac{n+m}{p} + \log(\max(n,m)))$ on p CPUs

merging by co-ranking:

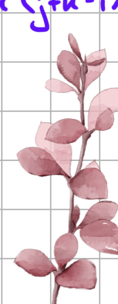
```
j = min(i,m);
k = i-j;
jlow = max(0,i-n);
klow = 0;
done = 0;

do {
  if (j>0&&k<n&&A[j-1]>B[k]) {
    d = (1+j-jlow)/2;
    klow = k;
    j -= d;
    k += d;
  } else if (k>0&&j<m&&B[k-1]>=A[j]) {
    d = (1+k-klow)/2;
    jlow = j;
    k -= d;
    j += d;
  } else done = 1;
} while (!done);
```

}co-ranks

"$O(\log(\max(n,m)))$ is not a fraction of $O(n+m)$. Therefore Ahmdal's Law does not apply"
— scriptum, p. 40

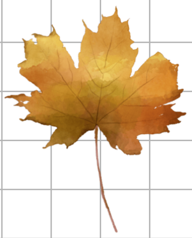
- Lemma: for i in range$(0, n+m)$ $\exists j \exists k$ $(j+k=i)$ with $j=0$ or $A[j-1] \le B[k]$ and $k=0$ or $B[k-1] \le A[j]$



- theorem: A and B can be merged work-optimally in $O(\frac{n+m}{p} + \log(\max(n,m)))$ on p CPUs with p processor-cores. The algorithm is stable and perfectly load-balanced.

bitonic merge*: oblivious merging algorithm, doesn't require concurrency
↳ sequence: $a_0, a_1, \dots, a_{n-1}$ with $n > 1$ comparable elements and $a_i \le a_j$ or $a_j \le a_i$ is a bitonic sequence if:
1. $\exists i$ with $0 \le i < n$ so that $a_0 \le a_1 \le \dots \le a_i$ and $a_{i+1} \ge a_{i+2} \ge \dots \ge a_{n-1}$
or 2. $\exists$ cyclic shift in sequnce
or 3. $n=1$

↳ Lemma: if $(a_{n-1})_{n \ge 1}$ is bitonic, then so are $\min(a_0, a_{\frac{n}{2}}), \min(a_1, a_{\frac{n}{2}+1}), \dots, \min(a_{\frac{n}{2}-1}, a_{n-1})$ and $\max(a_0, a_{\frac{n}{2}}), \max(a_1, a_{\frac{n}{2}+1}), \dots, \max(a_{\frac{n}{2}-1}, a_{n-1})$ of length $\frac{n}{2}$ and $\forall (\min(a_i, a_{\frac{n}{2}+i}))_{i \ge 0} \le (\max(a_i, a_{\frac{n}{2}+i}))_{i \ge 0}$ (of even length)



recursive bitonic ordering: 1. Split sequence in two bitonic halves

2. Order elements

↳ total work: $W(n) = \frac{n}{2}\log_2(n)$

↳ useful for merging: order A in incresing order, B in decreasing order

↳ sorts in $O(\log^2(n))$ parallel steps and $O(n\log(n))$ work

↳ commonly used in computer networks, sometimes on sorting networks

Prefix-sums Problem: compute the i-th prefix-sum for all i

↳ exclusive sum for $0 \le i < n$: $B[i] = \bigoplus_{j=0}^{i-1} A[j]$ ⇒ exscan

↳ inclusive sum for $0 \le i < n$: $B[i] = \bigoplus_{j=0}^{i} A[j]$ ⇒ scan

↳ generalization of reduction problem (compute $B[n-1] = \bigoplus_{j=0}^{n-1} A[j]$)

↳ sequential complexity: $O(n)$

agrees on common val. & distributes outcome to processors

Load Balancing with Prefix-sums ("parallel array computation"): solves load balancing problem by using an allreduce operation

1. foreach (a : A) {
   a = 0 for unmarked, a = 1 for marked elements
2. } perform prefix-sums computation for A on B
3. foreach (marked element) {
   B[i] = number of marked elements up to i
   store marked elements in new array with length m
4. } Partition element array into p blocks of size $\frac{m}{p}$

↳ Result: all processors have same amount of non-trivial work

divide-and-conquer

Recursive prefix-sums:

```
void Scan(int A[], int n)  A.length = n
{
  if (n==1) return;  A[0] = prefix-sum (no computation necessary)

  int B[n/2];
  int i;
                                  A'[i] = A[2i] ⊕ A[2i+1]
  for (i=0; i<n/2; i++) B[i] = A[2*i]+A[2*i+1];

  Scan(n/2,B);

  A[1] = B[0];
  for (i=1; i<n/2; i++) {
    A[2*i]   = B[i-1]+A[2*i];  B[2i] = B'[i-1] ⊕ A[2i]
    A[2*i+1] = B[i];  B[2i+1] = B'[i]
  }
  if (n%2==1) A[n-1] = B[n/2-1]+A[n-1];
}
```

i-th prefix-sum of A (even i):

$$\bigoplus_{j=0}^{i} A[i] = \bigoplus_{j=0}^{\lfloor i/2 \rfloor} (A[2j] \oplus A[2j+1]) \oplus A[i]$$

i-th prefix-sum of A (odd i):

$$\bigoplus_{j=0}^{i} A[i] = \bigoplus_{j=0}^{\lfloor i/2 \rfloor} (A[2j] \oplus A[2j+1])$$

Induction Hypothesis:

$B[i] = \bigoplus_{j=0}^{i} (A[2j] \oplus A[2j+1])$, then

$B[\lfloor i/2 \rfloor] = \bigoplus_{j=0}^{i} A[i]$ for odd i

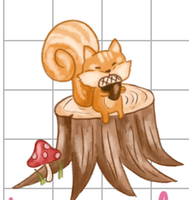
$B[\lfloor i/2 \rfloor - 1] \oplus A[i] = \bigoplus_{j=0}^{i} A[i]$ else

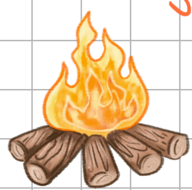
Parallel running time with p processors:

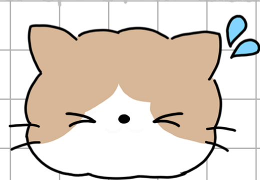
$O(\frac{n}{p} + \log(n))$

Master Theorem: solution to recurrence of form $T(n) = aT(\frac{n}{b}) + \Theta(n^d \log^e(n))$: $a \ge 1, b > 1, d \ge 0, e \ge 0, T(1) = \text{constant}$

1. if $(\frac{a}{b^d} < 1)$: $T(n) = \Theta(n^d \log^e(n))$
2. if $(\frac{a}{b^d} = 1)$: $T(n) = \Theta(n^d \log^{e+1}(n))$
3. if $(\frac{a}{b^d} > 1)$: $T(n) = \Theta(n^{\log_b(a)})$







Iterative prefix-sums:

1. iterat.: A[0]+A[1], A[2]+...
2. iterat.: A[0]+A[2], A[4]+A[6]
3. iterat.: A[0]+A[4], A[8]+A[12]

```
int k, kk;
int i;

// up-phase
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  for (i=kk-1; i<n; i+=kk) A[i] = A[i-k]+A[i];
}
// down-phase
for (k=k>>1; k>1; k=kk) {
  kk = k>>1; // halve
  for (i=k-1; i<n-kk; i+=k) A[i+kk] = A[i]+A[i+kk];
}
```

correctness can be proven by invariants

worse spacial locality than recursive version

theorem: trade-off for prefix-sum computation

$s + t \ge 2n - 2$ for s = size, t = depth

amount of ⊕ operations

Non work-optimal, faster prefix-sums:

```
int *a, *b, *t;
a = A; b = B;

k = 1;
while (k<n) {
  // update into B
  for (i=0; i<k; i++) b[i] = a[i];
  for (i=k; i<n; i++) b[i] = a[i-k]+a[i];
  k <<= 1; // double

  // swap
  t = a; a = b; b = t;
}
if (a!=A) for (i=0; i<n; i++) A[i] = B[i]; // copy back when necessary
```

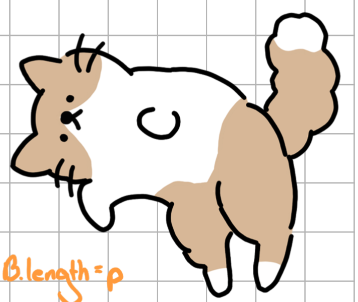
1. iteration: $A[i] = A[i-1] \oplus A[i]$
2. iteration: $A[i] = A[i-2] \oplus A[i]$
3. iteration: $A[i] = A[i-4] \oplus A[i]$

Reduced complexity of: $O(\frac{n \log(n)}{p} + \log(n))$

Correctness is easy to prove by invariants

Similar problem: list-ranking problem

Solution complexity: $O(\frac{n}{p} + \log(n))$ on EREW PRAM



Blocking: non-work optimal algorithms = useful as building blocks

1. int A[], A.length = n, p = number of processors
2. Divide A into p blocks of size $\frac{n}{p}$
3. Each CPU: perform seq. reduction on block and put results into int B[], B.length = p
4. Compute all prefix-sums of B
5. Each CPU: add B[i] to first element of A-block

→ Complexity: $O(\frac{n}{p} + \frac{p \log(p)}{p} + \frac{n}{p}) = O(\frac{n}{p} + \log(p))$

Careful Application of Blocking:

1. Divide input into p+1 blocks of size $\lceil \frac{n}{p+1} \rceil$ or $\lfloor \frac{n}{p+1} \rfloor$
   - → blocks are ordered, first block contains first $\frac{n}{p+1}$ elements, etc
   - → time t = number of $\oplus$ operations that have to be carried out sequentially
   - → work/size s = number of $\oplus$ operations carried out by p processors
2. For each block: compute inclusive prefix-sums for all block elements
   - → $t_1 = \frac{n}{p+1} - 1$, $s_1 = p(\frac{n}{p+1} - 1)$
3. Compute inclusive prefix-sums for the sequence of p sums
   - → $t_2 = p-1$, $s_2 = p-1$
4. For each block that isn't 0 or p: Add prefix-sum for the last block to the first $\frac{n}{p+1} - 1$ elements of the block
   - → $t_{3.1} = \frac{n}{p+1} - 1$, $s_{3.1} = (p-1)(\frac{n}{p+1} - 1)$
   - a) Last block p: Add prefix-sum for block p-1 to the first element
     - Compute inclusive prefix-sums for all elements of the bl.
     - → $t_{3.2} = t_{3.1} + 1$, $s_{3.2} = \frac{n}{p+1}$
   - → $t_3 = \frac{n}{p+1}$, $s_3 = 1 + p(\frac{n}{p+1} - 1)$

Total time: $t = 2(\frac{n}{p+1} - 1) + p$

Total work: $p = 2p(\frac{n}{p+1} - 1) + p$

Total sum: $s + t = 2n - 2$

Complexity: $O(\frac{n}{p} + p)$