



Parallel Sorting Algorithms

TDDD56 Lecture 6/7

Christoph Kessler

PELAB / IDA
Linköping university
Sweden

2015

TDDD56 Outline



Lecture 1: Multicore Architecture Concepts

Lecture 2: Parallel programming with threads and tasks

Lecture 3: Shared memory architecture concepts

Lecture 4: Non-blocking Synchronization

Lecture 5/6: Design and analysis of parallel algorithms

Lecture 6/7: Parallel Sorting Algorithms

Lecture 8: Loop parallelization and optimization

2

Motivation



- Sorting is one of the most important subroutines
- Large data sets,
little computation, mainly control and data movement
- Design of *scalable* parallel sorting algorithms is not straightforward
 - As opposed to many numerical kernel problems
- Parallel sorting algorithms heavily investigated since 40+ years
- Many parallel sorting algorithms known today
- Consider **3 representatives**, here for shared-memory:
 - Parallel Quicksort + variants
 - Bitonic Sort
 - Parallel Mergesort
- All based on parallel divide-and-conquer principle

3

Simple Parallel Quicksort



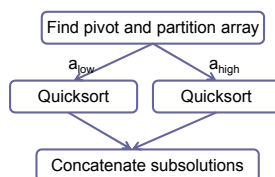
Recall: Sequential Quicksort



- Algorithmic design pattern: Divide-and-Conquer

```

void quicksort ( int a[n] )
{
    // divide phase:
    choose pivot a[i] for some i in 0...n-1;
    // partition array a into alow, ahigh:
    alow = { a[j] with a[j] <= a[i] };
    ahigh = { a[j] with a[j] >= a[i] };
    // recursive calls:
    quicksort ( alow );
    quicksort ( ahigh );
    // combine phase:
    a = concat ( alow, ahigh );
}
    
```



- Time complexity: $O(n \log n)$ on average, $O(n^2)$ worst-case

Remarks on Pivot choice



- First or last element in subsequence???
 - Worst-case time with pre-sorted data
- Pivot chosen randomly...
 - still not unlikely to generate unbalanced subproblems
- Better: draw random sample e.g. of size $O(\sqrt{n})$ and choose pivot as the median of these
 - improves balance of a_{low} to a_{high}
- Pivot randomly attached to one of the two partitions
 - Randomly to avoid continued disbalance,
 - Attachment avoids separate treatment, e.g. in concat

6

Remarks on Partitioning

■ In-place partitioning (reordering)

- See your algorithms course or textbook for proof of correctness

```

left = 0; right = n-1;
do {
    while ( a[left] < a[i] ) left++;
    while ( a[right] > a[i] ) right--;
    exchange ( a[left++], a[right--] );
} while ( left < right );
    
```



- ☺ Avoids separate arrays for a_{low} , a_{high}
- ☺ Pointers suffice, concat implicit
- ☺ Cache friendly (why?)

7

Background: Divide-and-Conquer

General strategy:

Divide-and-conquer(Problem P of size n):

If P is **trivial** (i.e., n very small), **solve** P directly.
otherwise:

divide P into $q \geq 1$ **independent** subproblems P_1, \dots, P_q of smaller size n_1, \dots, n_q

solve these recursively:

$S_1 \leftarrow \text{Divide-and-conquer}(P_1, n_1)$,

\vdots

$S_q \leftarrow \text{Divide-and-conquer}(P_q, n_q)$

combine the subsolutions S_1, \dots, S_q to a solution S for P .

8

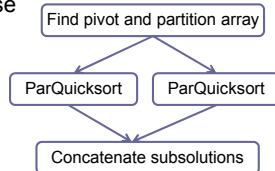
Parallel Divide-and-Conquer Simple Parallel Quicksort

- The q subproblems (of size $< n$) are **independent** of each other

- can execute the recursive calls in parallel
- defines q independent subtasks
- **SimpleParQuicksort** algorithm

- Achievable speedup is quite limited if the *divide* and the *combine* phase are not parallelized (cf. Amdahl's Law!)

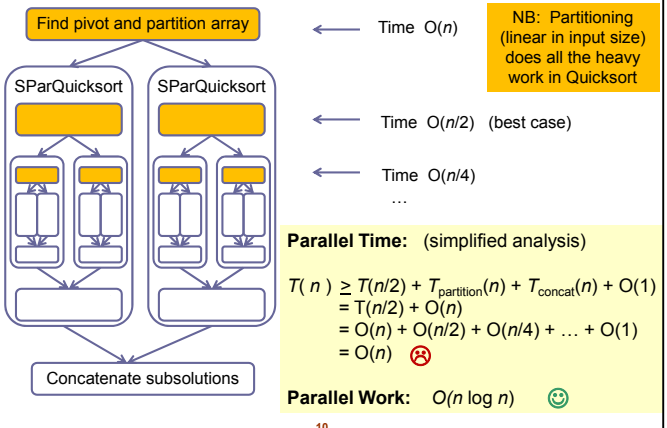
- Example:
Parallel Quicksort, general case:



$$T(n) \geq T(n/2) + T_{\text{partition}}(n) + T_{\text{concat}}(n) + O(1) = T(n/2) + O(n)$$

9

Simple Parallel Quicksort, Analysis



10

Simple Parallel Quicksort, Discussion

- Parallel time $\Omega(n)$
→ speedup $O(\log n)$ ☹
- ok for p in $O(\log n)$ i.e. for small processor count
- Task granularity control
 - Avoid too small tasks (overhead of spawning...)
 - Spawn SimpleParQuicksort (a_{high}) to different processor if size $> n/p$, otherwise do it sequentially
 - Parallel time: still $O(n)$
sequence of partitions, time $n + n/2 + n/4 + \dots = O(n)$
plus seq. sorts of size $\leq n/p$, time $O(n/p * \log(n/p))$

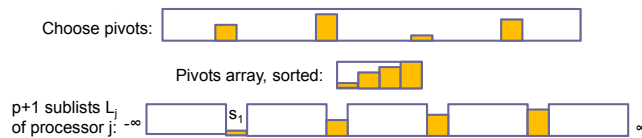
11

Parallel Sample Sort

How to partition in parallel?

Approach 1: parallel $(p+1)$ -way partitioning (Parallel Sample Sort)

- choose $p-1$ pivots (splitters) $s_1 \dots s_{p-1}$ from array a ($s_0 = -\infty, s_p = \infty$) and sort them (sequentially)
- each processor i partitions n/p elements of array a into p partial lists $L_{i,j}, j=0 \dots p-1$, according to pivots: $a[k]$ into list $L_{i,j}$ iff $s_j < a[k] < s_{j+1}$, for $k=i*n/p \dots (i+1)*n/p-1$
- each processor j gathers all partial lists $L_{i,j}$ into list L_j
- each processor j sorts its list L_j sequentially



Example for Parallel Samplesort

- Here, $p=3$

Input ($n=15$): 9, 3, 17, 4, 5, 20, 19, 11, 1, 8, 7, 2, 15, 14, 6
 Pivot candidates: 17, 19, 7, 6
 Choose $p-1$ pivots and sort them: 7, 17

Proc0: list $L_{0,0}=(3,4,5)$ $L_{0,1}=(9)$ $L_{0,2}=(17)$
 Proc1: list $L_{1,0}=(1)$ $L_{1,1}=(8,11)$ $L_{1,2}=(19,20)$
 Proc2: list $L_{2,0}=(2,6,7)$ $L_{2,1}=(14,15)$ $L_{2,2}=-$

Proc0 sorts: 3,4,5,1,2,6,7
 Proc1 sorts: 9,8,11,14,15
 Proc2 sorts: 17,19,20

14

Parallel Sample Sort, Analysis

- Sequential sorting of the $p-1$ pivots: time $O(p \log p)$
- Parallel $(p-1)$ -way-partitioning in time $O((n/p) * \log p)$ because of binary search in the sorted Pivots array
- Simultaneous sequential sorting of p lists L_j in time $O(n/p * \log(n/p))$
- Total: parallel time $O((n/p + p) * \log(n/p + p))$
 - time-optimal for p in $O(n/p)$, i.e. p in $O(\sqrt{n})$.
- Advantages:
 - no recursive calls
 - can also be used on message-passing machines (one all-to-all communication)
- Disadvantage: not in-place, lists $L_{i,j}$ need separate array

Fully Parallel Quicksort

How to partition in parallel? (2)

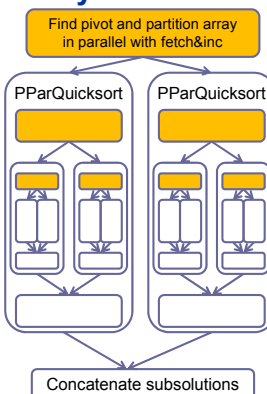
Approach 2: Using Atomic Fetch&Increment

- If the hardware provides it ...
- Keep degree-2 divide-and-conquer, parallelize partition loop
- Each processor partitions part of array of size n/p . Then re-order partial partitions
- Needs a temporary array

```
ParPartition( int a[n] )
{
    shared int b[n]; // temp. array
    shared int left = 0, right = n-1; // shared counters
    forall i in 0...n-1 do in parallel
        if ( a[i] <= pivot ) b[ fetch&inc( left, +1 ) ] = a[i];
        else b[ fetch&inc( right, -1 ) ] = a[i];
    // ... and copy back b to a in parallel for in-place partitioning
}
```

17

Fully Parallel Quicksort with Fetch&Inc, Analysis



Time $O(n/p)$ on p processors, up to $p \leq n$ procs could be used!

Time $O(n/p)$ (best case) on p processors

Time $O(n/p)$

Parallel Time: (simplified analysis)

$$\begin{aligned}
 T(n) &\geq T(n/2) + T_{\text{ParPartition}}(n) + T_{\text{concat}}(n) + O(1) \\
 &= T(n/2) + O(n/p) \\
 &= O(n/p) + O(n/p) + \dots + O(n/p) \\
 &= O((n \log n) / p) \quad \text{😊}
 \end{aligned}$$

Parallel Work: $O(n \log n)$ 😊

18

Fully Parallel Quicksort with Fetch&Inc, Remarks



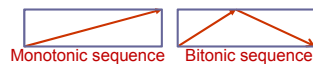
- Could utilize up to n processors in parallel on each level, then have $O(n/p)$ i.e. constant time per level ☺
 - IF the hardware implementation does **not serialize** all Fetch&Inc ops!
 - ▶ p concurrent calls to Fetch&Inc done per level
 - Not a realistic assumption for large p
 - Requires a Combining CRCW PRAM (i.e., a very strong shared memory interface)
 - Only few parallel systems (such as SBPRAM) support this.
- Only applicable if the hardware provides atomic fetch-and-inc
 - Software implementation of f&i with mutex lock would serialize anyway
- **Alternative to hardware Fetch&Inc:**
Implement scalable parallel Fetch&Inc over p processors in software with *Parallel Prefix Sums* (see Lecture 6)
 - Adds factor $\log p$ to the time complexity: $O(n/p \log n \log p)$, i.e., expected parallel time $O((\log n)^2)$ with n processors

19



Bitonic Sort

Bitonic Sort



- A sequence of numbers $a=(a_1, \dots, a_n)$ is called *bitonic* if either there is a k in $\{1, \dots, n\}$ such that $a_1 \leq \dots \leq a_k \geq \dots \geq a_n$ or the sequence can be rotated to that form

21

Example for Bitonic Sort



- $a = (4, 1, 2, 7, 6, 5)$ is a bitonic sequence
(could be rotated left to $1, 2, 7, 6, 5, 4$)

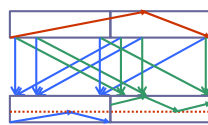
 $k=3$, $a_3=7$ is largest element, $1, 2, 7$ is ascending part,
 $6, 5, 4$ is descending part
- $4, 1, 7, 6, 5, 2$ is not a bitonic sequence
(cannot be rotated to have a unique „peak“)
 $1, 7, 6, 5, 2, 4$ $7, 6, 5, 2, 4, 1$
 $6, 5, 2, 4, 1, 7$ $5, 2, 4, 1, 7, 6$ $2, 4, 1, 7, 6, 5$

22

Bitonic Sort (cont.)



- A sequence of numbers $a=(a_1, \dots, a_n)$ is called *bitonic* if either there is a k in $\{1, \dots, n\}$ such that $a_1 \leq \dots \leq a_k \geq \dots \geq a_n$ or the sequence can be rotated to that form
- **Lemma** (Batcher, 1968):
If a is bitonic, then
 $a' = \min(a_1, a_{n/2+1}), \dots, \min(a_{n/2}, a_n)$
 $a'' = \max(a_1, a_{n/2+1}), \dots, \max(a_{n/2}, a_n)$
 are both bitonic and $\max(a') \leq \min(a'')$
- Kind of divide step for bitonic sequences



Proof not shown here, see e.g. Cormen et al.

23

Example for Bitonic Sort



- $a = (4, 1, 2, 7, 6, 5)$ is a bitonic sequence
(could be rotated left to $1, 2, 7, 6, 5, 4$)

 $k=3$, $a_3=7$ is largest element, $1, 2, 7$ is ascending part,
 $6, 5, 4$ is descending part
- $4, 1, 7, 6, 5, 2$ is not a bitonic sequence
(cannot be rotated to have a unique „peak“)
 $1, 7, 6, 5, 2, 4$ $7, 6, 5, 2, 4, 1$
 $6, 5, 2, 4, 1, 7$ $5, 2, 4, 1, 7, 6$ $2, 4, 1, 7, 6, 5$
- Batcher's lemma for $a = (4, 1, 2, 7, 6, 5)$ from above:
 $a' = \min(4, 7), \min(1, 6), \min(2, 5) = 4, 1, 2$ is bitonic
 $a'' = \max(4, 7), \max(1, 6), \max(2, 5) = 7, 6, 5$ is bitonic

24

BitonicMerge

- Consequence of Batcher's Lemma:

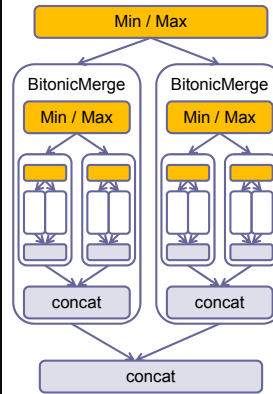
```

BitonicMerge ( int a[n], int order )
{ // a must be bitonic
  compute a', a'' according to Lemma if order == ascending
  (exchange min and max if order == descending)
  return concat ( BitonicMerge( a', order),
                  BitonicMerge( a'', order) )
} // now sorted in order
    
```

Parallel divide+conquer,
base case is $n=2$
(trivial to merge)

- Analysis:
bitonic sequence can be b.-merged in parallel time $O(\log n)$
with n processors ($n/2$ doing min, $n/2$ doing max)
- Note: ascending/descending order needed in a minute

Bitonic Merge, Analysis



Parallel Time:

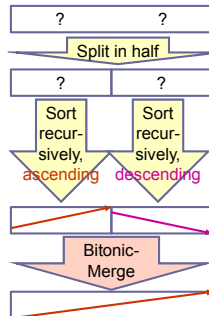
$$\begin{aligned}
 T_{\text{BitonicMerge}}(n) &= T_{\text{BitonicMerge}}(n/2) + T_{\text{MinMax}}(n) \\
 &= T_{\text{BitonicMerge}}(n/2) + O(1) \\
 &= O(1) + O(1) + \dots + O(1) \\
 &= O(\log n)
 \end{aligned}$$

Parallel Work:

$$\begin{aligned}
 W_{\text{BitonicMerge}}(n) &= 2 W_{\text{BitonicMerge}}(n/2) + O(n) \\
 &= \dots \\
 &= O(n \log n)
 \end{aligned}$$

BitonicMerge – and then?

- BitonicMerge brings any bitonic sequence into sorted form.
- But how do we derive, from unsorted input data, a bitonic sequence?
- Idea: combine both problems, sort recursively and flip the order
- Parallel Divide-and-Conquer, again!



Bitonic Sort

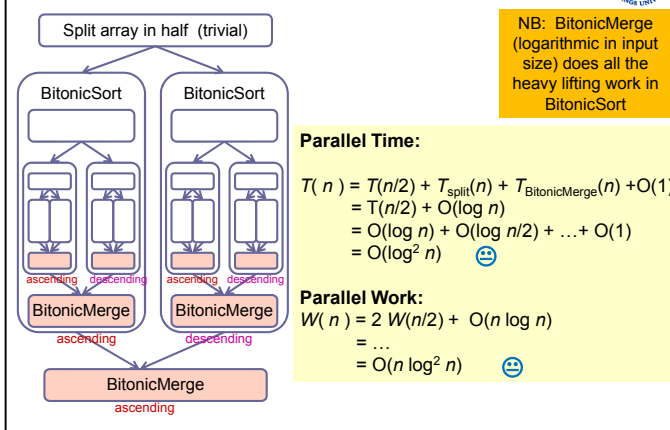
- Idea: Turn arbitrary sequence into bitonic sequence by sorting its halves in ascending and descending order:

```

BitonicSort ( int a[n], int order )
{ // a is an arbitrary sequence
  BitonicSort( a[1..n/2], ascending );
  || BitonicSort( a[n/2+1..n], descending ); // in parallel
  // a is now bitonic
  BitonicMerge ( a, order );
}
    
```

- Analysis for n processors:
 $T(n) = T(n/2) + O(\log n) = O((\log n)^2)$
- Not time-optimal but the constant factor is very small

Bitonic Sort, Analysis



NB: BitonicMerge
(logarithmic in input
size) does all the
heavy lifting work in
BitonicSort

Parallel Time:

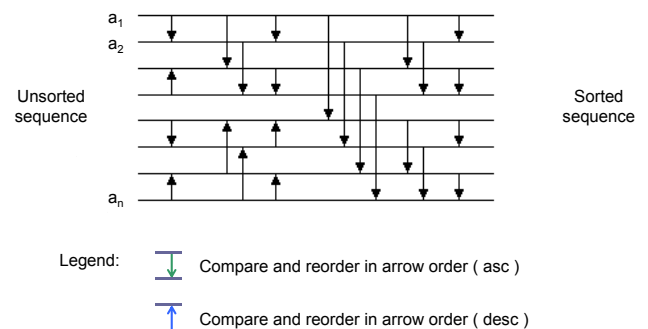
$$\begin{aligned}
 T(n) &= T(n/2) + T_{\text{split}}(n) + T_{\text{BitonicMerge}}(n) + O(1) \\
 &= T(n/2) + O(\log n) \\
 &= O(\log n) + O(\log n/2) + \dots + O(1) \\
 &= O(\log^2 n)
 \end{aligned}$$

Parallel Work:

$$\begin{aligned}
 W(n) &= 2 W(n/2) + O(n \log n) \\
 &= \dots \\
 &= O(n \log^2 n)
 \end{aligned}$$

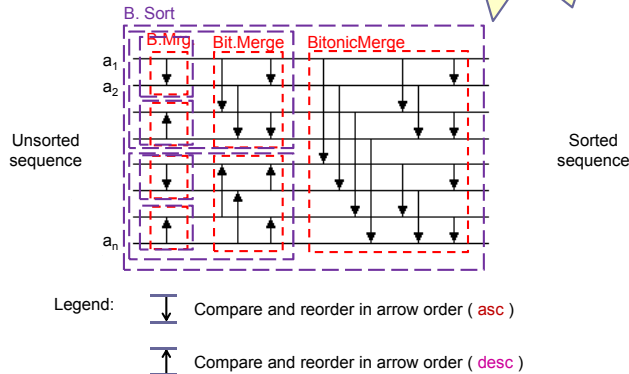
Bitonic Sort

- Unfolded task graph for $n=8$:



Bitonic Sort

- Unfolded task graph for $n=8$:



31

Remarks on Bitonic Sort

- Bitonic Sort is a **sorting network**, designed as massively parallel hardware algorithm (comparator network) from the beginning
- Bitonic Sort is an **oblivious algorithm**, i.e., control flow only depends on input *size*, not on input *contents*
 - Stable, well predictable performance, good for realtime computing
- Add granularity control:
 - Stop BitonicSort recursion at problem size n/p and sort sequentially instead

32

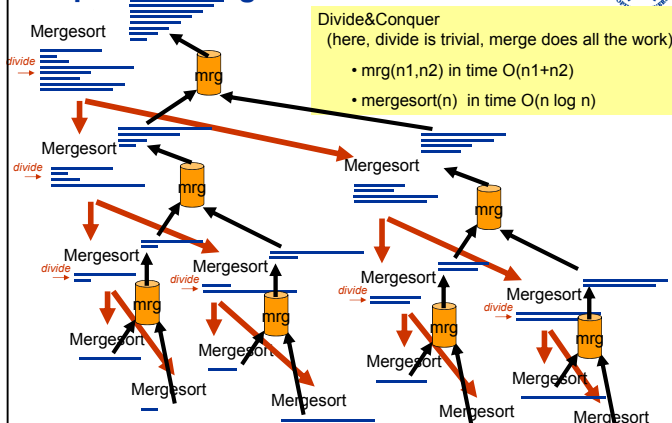
Parallel Mergesort

Mergesort (1)

- Known from sequential algorithm design
 - Merge**: take two sorted blocks of length k and combine into one sorted block of length $2k$
- ```
SeqMerge (int a[k], int b[k], int c[2k])
{
 int ap=0, bp=0, cp=0;
 while (cp < 2k) { // assume a[k] = b[k] = ∞
 if (a[ap] < b[bp]) c[cp++] = a[ap++];
 else c[cp++] = b[bp++];
 }
}
```
- Sequential time:  $O(k)$
  - Can also be formulated for in-place merging (copy back)

34

## Sequential Mergesort



35

## Sequential Mergesort

**Time:  $O(n \log n)$**

```
void SeqMergesort (int *array, int n) // in place
{
 if (n==1) return;
 // divide and conquer:
 SeqMergesort (array, n/2);
 SeqMergesort (array + n/2, n-n/2);
 // now the subarrays are sorted
 SeqMerge (array, n/2, n-n/2);
}

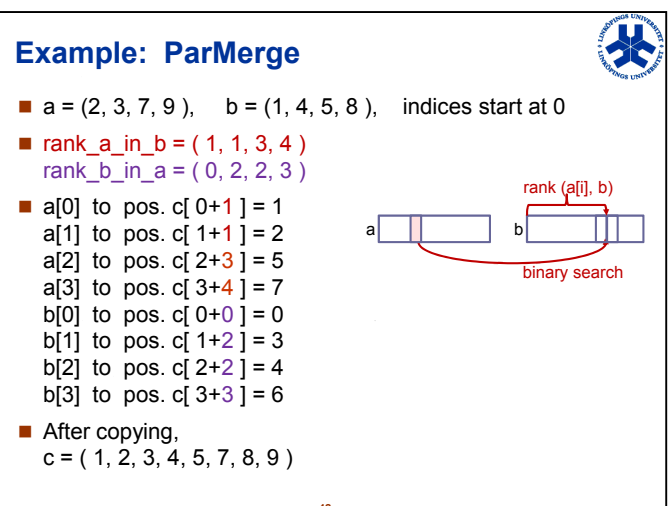
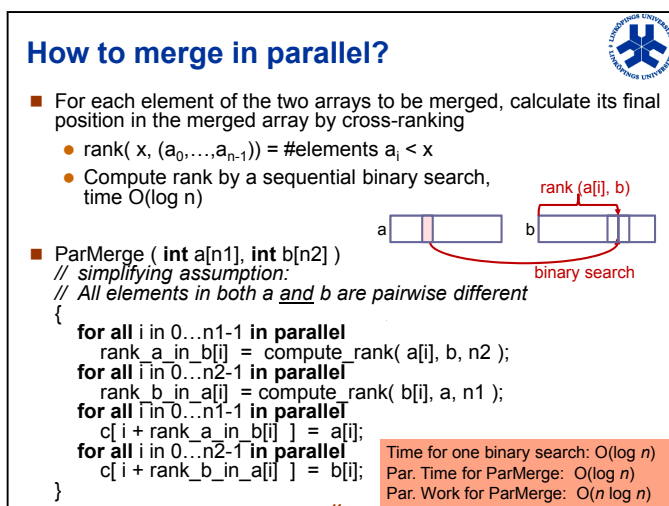
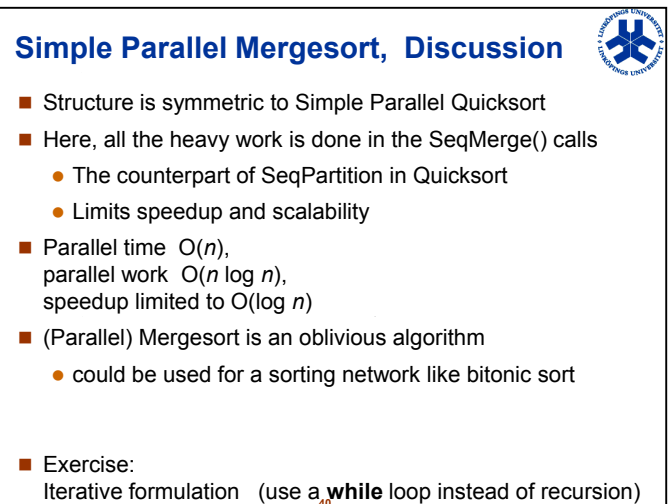
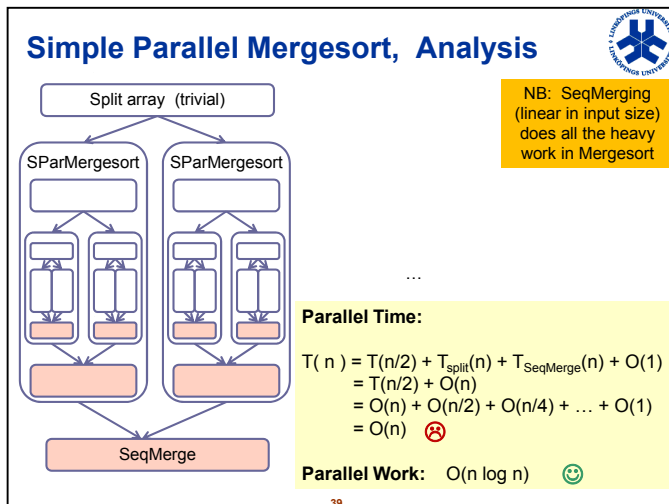
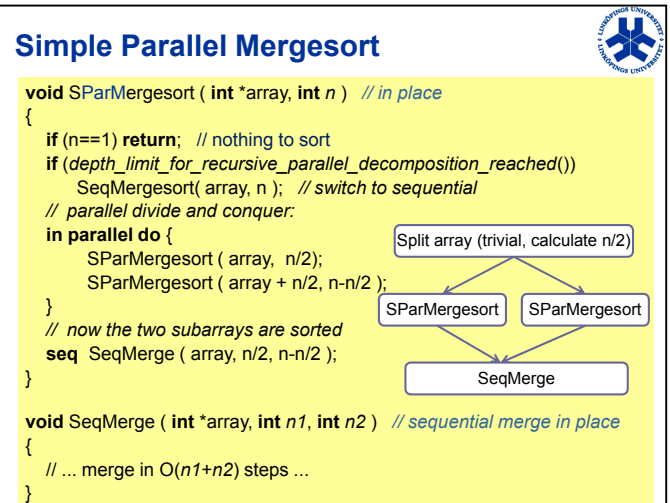
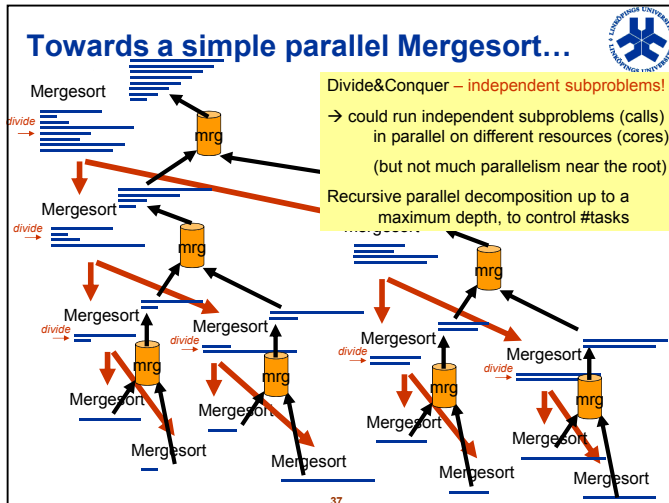
void SeqMerge (int array, int n1, int n2) // sequential merge in place
{
 ... ordinary 2-to-1 merge in $O(n1+n2)$ steps ...
}
```

Split array (trivial, calculate  $n/2$ )

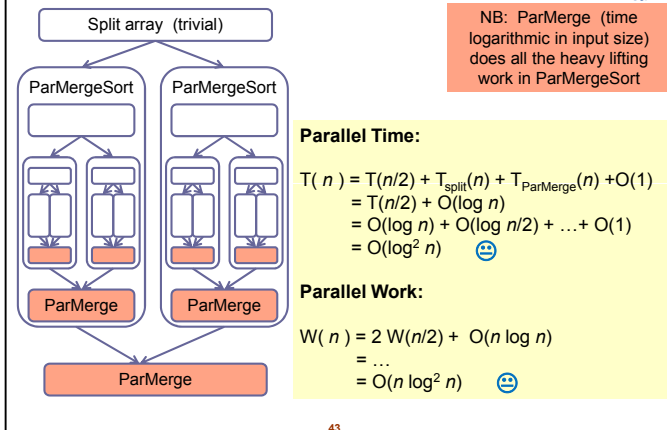
SeqMergesort   SeqMergesort

SeqMerge

36



## Fully Parallel Mergesort, Analysis



43

## Summary – Parallel Sorting



- We considered a few common parallel sorting algorithms
  - Simple Parallel Quicksort
  - Parallel Samplesort
  - Fully Parallel Quicksort with Fetch&Inc / with ParallelPrefixSums
  - Bitonic Sort
  - Simple Parallel Mergesort
  - Fully Parallel Mergesort
- Many more exist:
  - e.g. parallel rank sort, parallel radix sort, ...
- There exist time-optimal ( $O(\log n)$ ) parallel sorting algorithms for  $n$  processors, but these are very complex
  - AKS-network (1981), Cole's pipelined parallel mergesort (1988)
- Algorithm engineering necessary to adapt textbook algorithms to perform efficiently on real parallel systems
  - Example: use SIMD hardware, run on GPUs, on distributed memory, on special network topologies, ...

## Self-Assessment: Fill in the table!



| Parallel Sorting Algorithm             | Uses up to how many procs | Parallel time | Parallel work | Parallel cost | Restrictions |
|----------------------------------------|---------------------------|---------------|---------------|---------------|--------------|
| Simple parallel quicksort              |                           |               |               |               |              |
| Parallel samplesort                    |                           |               |               |               |              |
| Fully parallel quicksort, f&i          |                           |               |               |               |              |
| Fully par. quicks. with par. prefix s. |                           |               |               |               |              |
| Bitonic sort                           |                           |               |               |               |              |
| Simple parallel mergesort              |                           |               |               |               |              |
| Fully parallel mergesort               |                           |               |               |               |              |

45

Questions?

## Further Reading



- J. Keller, C. Kessler, J. Träff: **Practical PRAM Programming**. Wiley Interscience, New York, 2001.
- J. JaJa: **An introduction to parallel algorithms**. Addison-Wesley, 1992.
- D. Cormen, C. Leiserson, R. Rivest: **Introduction to Algorithms**, Chapter 30. MIT press, 1989.
- H. Jordan, G. Alaghband: **Fundamentals of Parallel Processing**. Prentice Hall, 2003.
- W. Hillis, G. Steele: Data parallel algorithms. *Comm. ACM* **29**(12), Dec. 1986. Link on course homepage.
- Fork compiler with PRAM simulator and system tools <http://www.ida.liu.se/chrke/fork> (for Solaris and Linux)

47

## Acknowledgements



- Some material courtesy of Jörg Keller, FernUniv. Hagen

48