Introduction to Parallel Computing

Algorithm examples: Merging, Prefix sums

Jesper Larsson Träff
traff@par.tuwien.ac.at
TU Wien
Parallel Computing

©Jesper Larsson Träff

First parallel algorithm: Merging

Problem:
Two ordered arrays A and B of size n and m, $A[i]\leq A[i+1]$ for $0\leq i<n-1$, and $B[j]\leq B[j+1]$ for $0\leq j<m-1$.
Merge A and B into array C of size n+m, $C[i]\leq C[i+1]$ for $0\leq i<n+m-1$, and each C[i] is either from A or B, and any element of A and B is in C

Useful building block in many other algorithms, e.g., mergesort

©Jesper Larsson Träff

<u>Problem</u>:
Two ordered arrays A and B of size n and m, $A[i]\leq A[i+1]$ for $0\leq i<n-1$, and $B[j]\leq B[j+1]$ for $0\leq j<m-1$.
Merge A and B into array C of size n+m, $C[i]\leq C[i+1]$ for $0\leq i<n+m-1$, and each C[i] is either from A or B, and any element of A and B is in C

Note:

- Sometimes helpful to assume that all elements from A and B are distinct.
- This assumption is without loss of generality (wlog): Make elements distinct by ordering pairs (A[i],i) lexicographically: $(A[i],i)<(A[j],j)$ if either $A[i]<A[j]$, or $A[i]=A[j]$ and $i<j$

Drawback:
This textbook trick can cost an extra n-element integer array (of original indices of the input elements), and extra time for comparison

Terminology/property:

Merge is stable, if the original (index) order of equal input elements in A and B is preserved in C, and if A[i]=B[j] for some i and j, then A[i] comes before B[j] in C

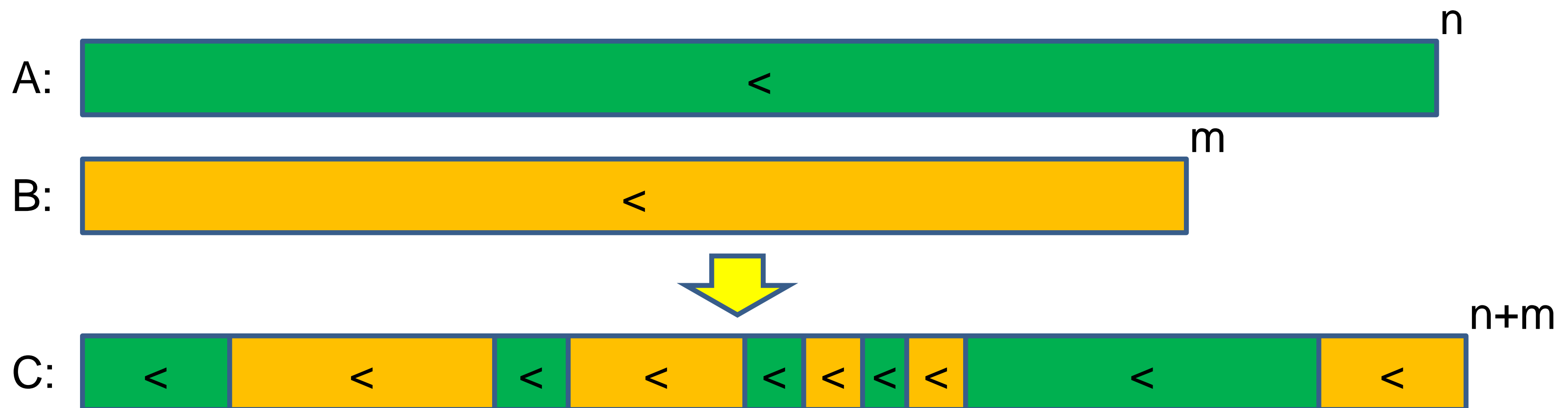
Stability is often desirable, and sometimes required (e.g., Radix sort)

NB: Not “best known” sequential implementation (See Knuth, Vol. 3 and many more recent papers: cache, branch avoidance, …)

Standard, strictly sequential solution:

```
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++];
```

Tseq(n+m) = O(n+m), m+n element reads and writes



Parallel solution?

```
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++];
```

No obvious parallelism in the sequential algorithm: Each iteration of the loop depends on what happened in previous iterations (i and j), and this depends on the input

No hope for "automatic parallelization"?

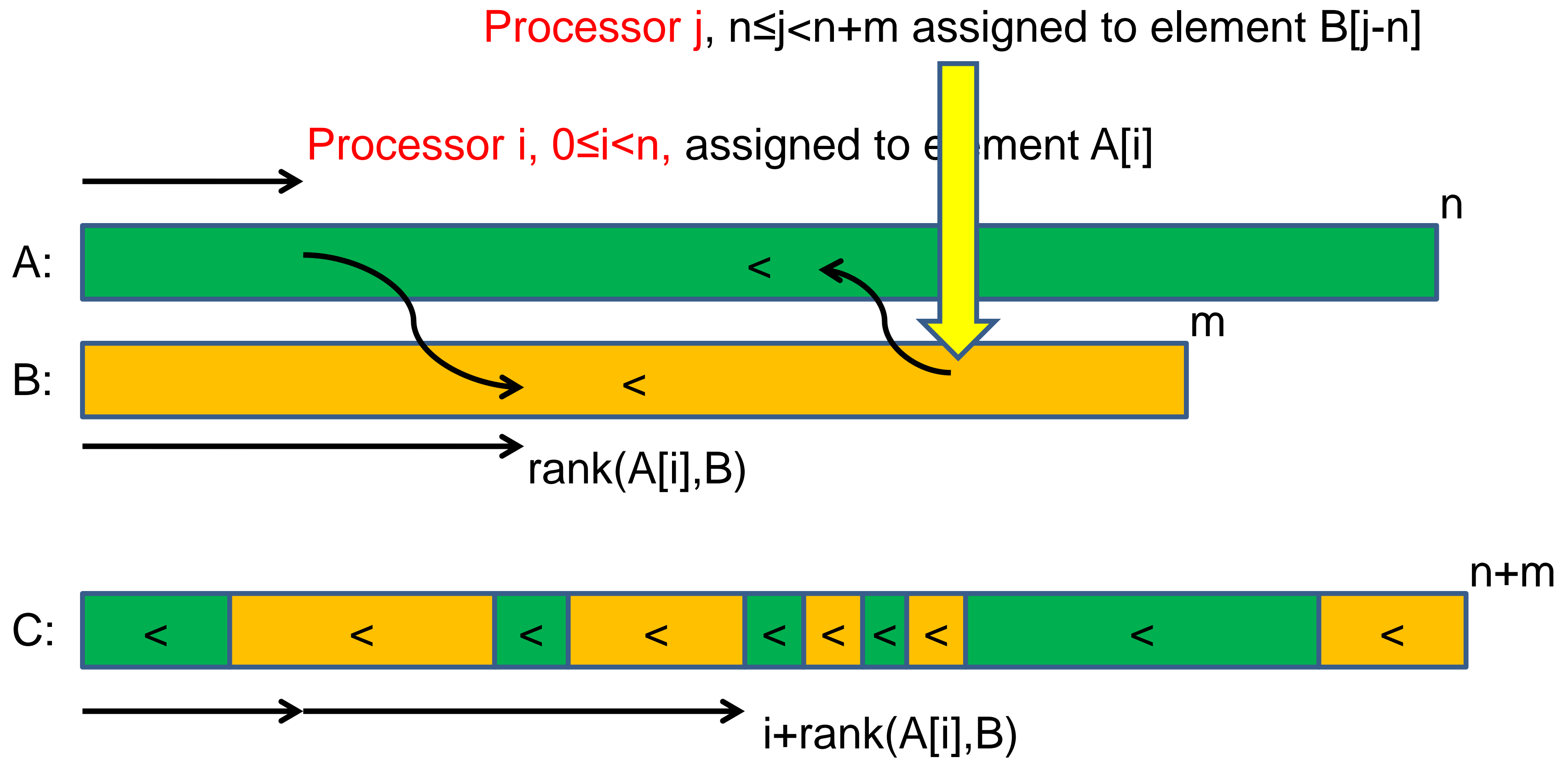
Parallelization approach: Divide input arrays into smaller parts of more or less the same size that can be merged independently. But how?

Parallel solution:
We try without specific assumptions about the parallel architecture. For now, just p processors that can access (part of) the input and output arrays

Solution 1:
Assume p=n+m processors, as many as elements in the input array. Assign a processor to each element in A and B

Definition:
Given an element x of a set A not containing x, rank(x,A) is the number of elements in A that are smaller than x

Idea: Merging by ranking

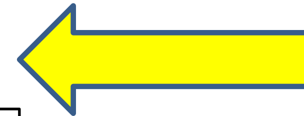


- Element A[i] is written to C[i+rank(A[i],B)]
- Element B[j] is written to C[j+rank(B[j],A)]
- All elements can be handled independently, in parallel

Parallel algorithm (and implementation) 1:

Processor i, 0≤i<n+m

```
if (i<n) C[i+rank(A[i],B)] = A[i];
else if (i<n+m) {
  j = i-n;
  C[j+rank(B[j],A)] = B[j];
}
```



Explicit parallelization: We specify what each processor has to do

Assumes access to input arrays A and B such that ranks can be computed efficiently. Convenient to have A, B, C in shared memory.

If not possible (distributed memory system), communication is needed for rankings and accessing elements

Observation:
For an ordered sequence stored in an array A, rank(x,A) can be computed sequentially by binary search. The number of operations per x (work, time) is O(log n) for input n-element array A

Tpar(n+m,n+m) = O(log (max(m,n))

Exponential improvement in time, with linear number of processors

Wpar(n+m,n+m) = O((m+n)log(max(n,m)) ≤ O(2n log n) = O(n log n)

The algorithm is not work-optimal:

$S_p(n) = O(n/(n \log n)/p) = O(p/\log n)$

Bad! Speed-up decreases with n

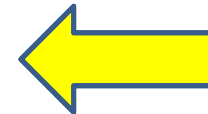
Problems:

- Algorithm is not efficient, work a factor (log n) too large
- Normally, n»p

- When is the computation finished (processors synchronized)?

Processor i, 0≤i<n+m

```
if (i<n) C[i+rank(A[i],B)] = B[i];
else if (i<n+m) {
  j = i-n;
  C[j+rank(B[j],A)] = B[j];
}
barrier; // synchronization construct
```



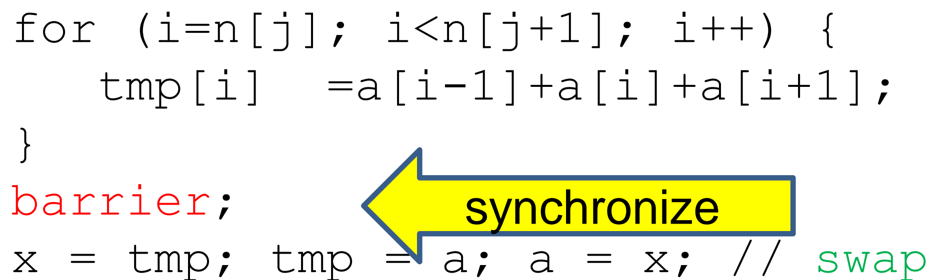
Explicit synchronization needed to ensure that any processor can access any C[k] element. Now merge finished

Barrier synchronization pattern (from slide set on par. patterns)

Logical processor synchronization:

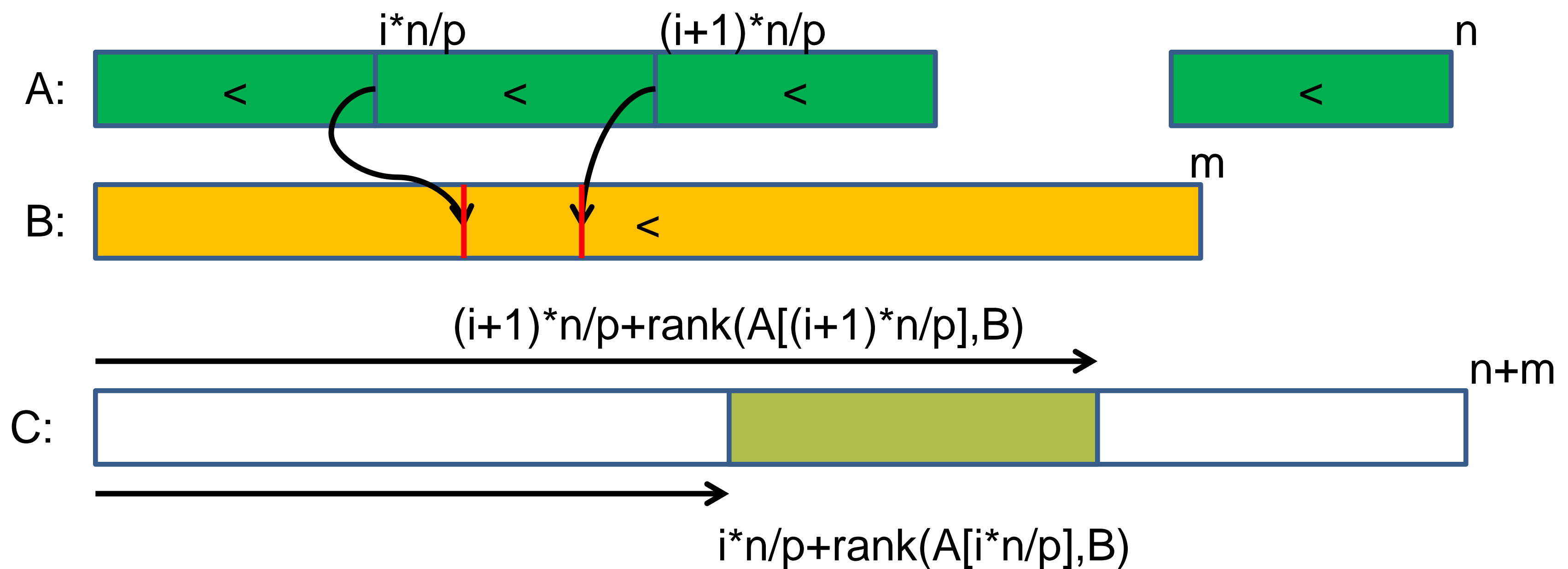
At <u>synchronization point</u> (code, instruction, …) no processor i can continue before all other processors j have also reached synchronization point. Different programming models have different ways of expressing barriers

Processor j, $0 \le j < p$



For correctness, synchronization needed. Can be explicit (barrier), implicit, or provided by programming or architecture model

Solution 2 (use fewer processors, some fixed p):
Divide A into p blocks of size approx. n/p, rank only first element of each block, in parallel merge blocks of A with blocks of B sequentially



©Jesper Larsson Träff

Processor i, 0≤i<p

Explicit parallelization

```
merge(&A[i*(n/p)],n/p,
      &B[rank(A[i*(n/p)],B)],
      rank(A[(i+1)*(n/p)],B)-rank(A[i*(n/p)],B),
      &C[i*(n/p)+rank(A[i*(n/p)],B)]);
barrier;
```

`merge(A,n,B,m,C)`: (sequentially) merges A of size n and B of size m into C of size n+m

Structure:

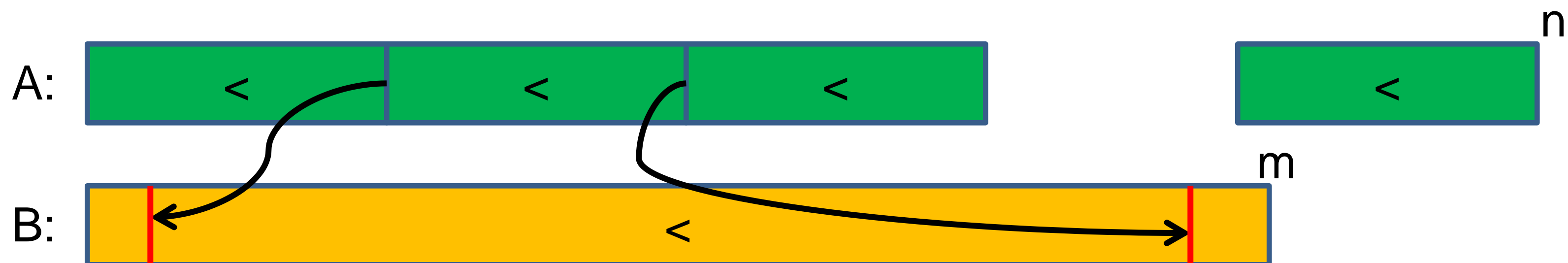
- Parallel preprocessing – rank: binary search - to divide problem into p independent pieces
- Sequential algorithm to process subproblems in parallel

Work optimal (for $p \le (m+n)/\log m$):

$W_{par}(p,n) = O(p \log m + p*(n/p)+m) = O(p \log m + (n+m)) = O(n+m)$

Problems:

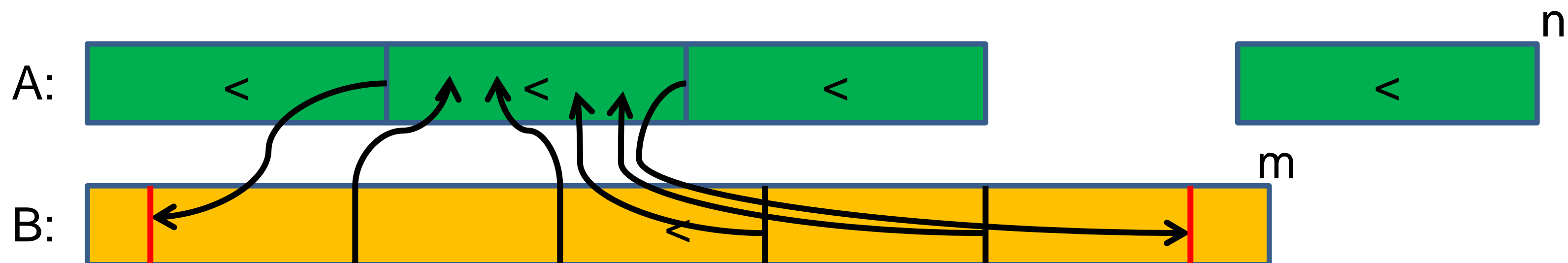
- Assumed that p divides n (can be fixed)
- Severe load imbalance in worst case



One processor does almost all work O(n/p+m), time Tpar(p,n+m) is O(n/p+m+log n)

<u>Solution 3a</u> (fix load imbalance):

Divide A into p blocks of size approx. n/p, rank only first element of each block. For all balanced pairs (good segments) where $\mathrm{rank}(A[(i+1)*n/p],B)-\mathrm{rank}(A[i*n/p],B)\leq m/p$, do sequential merge. For unbalanced pairs, divide bad segments $B[\mathrm{rank}(A[i*n/p],B),\ldots,\mathrm{rank}(A[(i+1)*n/p],B)[$ into smaller parts and rank first elements in A. Merge resulting pairs



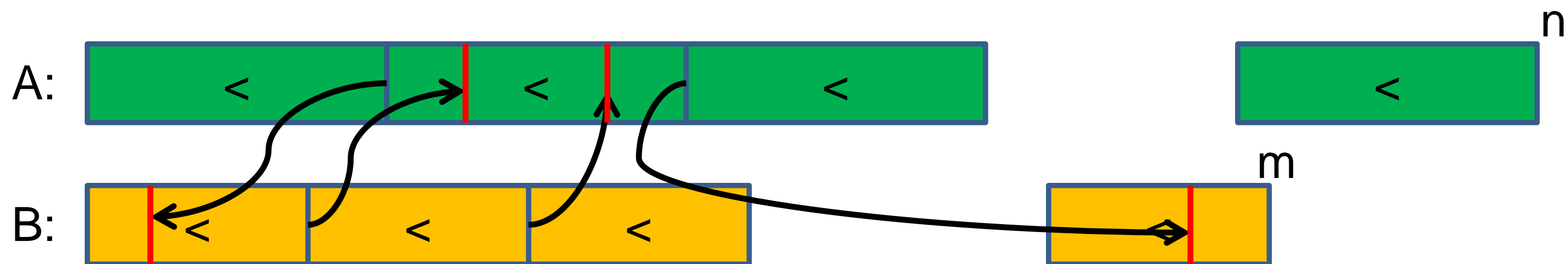
Bad segment,
$\mathrm{rank}(A[(i+1)*n/p],B)-\mathrm{rank}(A[i*n/p],B)>m/p$

Now at most 2p smaller merge problems, all of size $O(n/p+m/p)$. Load balance achieved

Problems:

- Assigning processors to indices i*(n/p) for $0 \le i < p$ easy.
- What about re-assigning to the start-indices of the blocks of the bad segments?
- What if there is more than one bad segment? Load balance must be done so that all blocks in bad segments have size at most m/p. This load balancing problem can be solved with prefix-sums (see later)

<u>Solution 3b</u> (avoid load imbalance):
Divide A into p blocks of size approx. n/p, rank only first element of each block. Divide B into p blocks of size approx. m/p, rank only first element of each block. This gives 2p merge pairs, all of size O(n/p+m/p); case analysis shows that they are independent. Merge resulting pairs sequentially in parallel



Torben Hagerup, Christine Rüb: Optimal Merging and Sorting on the EREW PRAM. Inf. Process. Lett. 33(4): 181-185 (1989)]
Jesper Larsson Träff: Simplified, stable parallel merging. CoRR abs/1202.6575 (2012)

Solution 4 (turning upside-down, merging by co-ranking):
Assume that for any given index i in the output array C, the (unique) two indices j and k in the input arrays A and B such that

C[0,…i-1] = merge(A[0,…j-1],j,B[0,k-1],k)

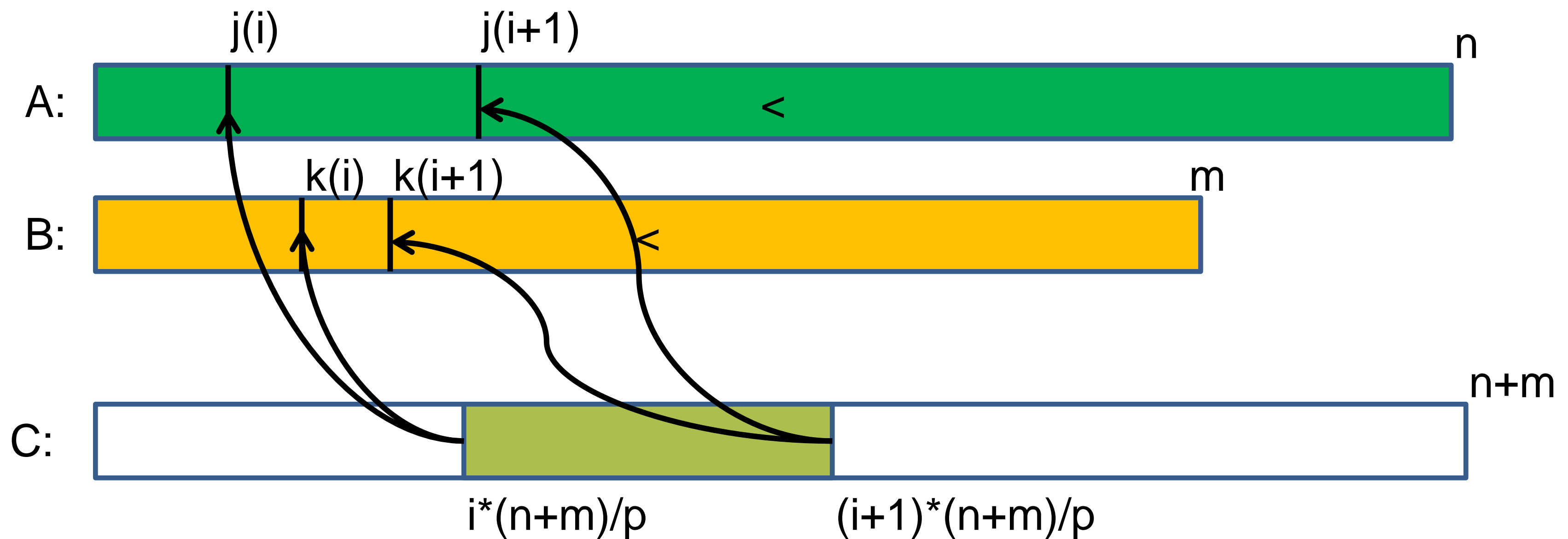
can be determined. Call j and k the co-ranks of i.

Divide the output array C of size n+m into p blocks of size (n+m)/p. The start index of block i, $0 \leq i < p$, is i*(n+m)/p.

For each block i:

- determine the co-ranks j(i) and k(i)
- merge the subsequences A[j(i),j(i+1)-1] and B[k(i),k(i+1)-1] into C[i*(n+m)/p,(i+1)*(n+m)/p-1]

Co-ranks of i*(n+m)/p are j(i), k(i), and satisfy j(i)+k(i)=i*(n+m)/p



Processor i, $0 \le i < p$

```
// coj[]: array of j-coranks
// cok[]: array of k-coranks
corank(i*(n+m)/p,A,n,&coj[i],B,m,&cok[i]);
barrier; // processor i will need coranks of i+1
merge(&A[coj[i]],coj[i+1]-coj[i],
      &B[cok[i]],cok[i+1]-cok[i],
      &C[i*(n+m)/p]); // sequentially
barrier;
```

Clearly work-optimal:

Wpar(p,n+m) ≈ p O((m+n)/p+log(n+m)) = O(m+n+p log(n+m))

which is O(m+n) when p log(n+m) in O(m+n)

Processor i, 0≤i<p

```
// coj[]: array of j-coranks
// cok[]: array of k-coranks
corank(i*(n+m)/p,A,n,&coj[i],B,m,&cok[i]);
barrier; // processor i will need coranks of i+1
merge(&A[coj[i]],coj[i+1]-coj[i],
      &B[cok[i]],cok[i+1]-cok[i],
      &C[i*(n+m)/p]); // sequentially
barrier;
```

Perfectly load balanced:
The co-ranking assumption helps to determine exactly the segments of A and B needed to merge the part C[i*(n+m)/p,…,(i+1)*(n+m)/p-1], and each processor handles a segment of C of the same size (difference at most 1 element if p does not divide (n+m))

Processor i, $0 \le i < p$

```
corank(i*(n+m)/p,A,n,&j1,B,m,&k1);
corank((i+1)*(n+m)/p,A,n,&j2,B,m,&k2);
merge(&A[j1],j2-j1,
      &B[k1],k2-k1,
      &C[i*(n+m)/p]); // sequentially
barrier;
```

"Synchronization-free, perfectly load-balanced, stable parallel merge"

Previous implementation needs synchronization (and communication?) after co-ranking, because processor i needs a result computed by processor i+1

Tradeoff: At the cost of a redundant co-rank computation, this synchronization step can be avoided. Which is better?

How can co-ranks be computed?

First: Observe that j+k=i (this will be an invariant): j and the k are the number of elements from A and B needed to produce the first i elements of C

Second: Let C[i-1] be the i'th output element of the merge, and let j and k be the co-ranks of i. Since C=merge(A,B), it holds that C[i'-1]≤C[i-1]≤C[i''-1] for any i',i'' with i'<i<i''

- Both A[j-1] and B[k-1] are in C[0,…,i-1], and the last element C[i-1] must be either A[j-1] or B[k-1].
- Neither A[j] nor B[k] are in C[0,…,i-1].

Doing the case analysis:

- $C[i-1]=A[j-1]$ implies $A[j-1]\leq B[k]$, and since $A[j-1]\leq A[j]$, trivially $B[k-1]<A[j]$ because $B[k-1]=C[i'-1]$ for some $i'<i$ and $B[k-1]<A[j-1]$.
- $C[i-1]=B[k-1]$ implies $B[k-1]<A[j]$, and trivially $A[j-1]\leq B[k]$.
- Therefore, for j and k to be co-ranks of i, both $A[j-1]\leq B[k]$ and $B[k-1]<A[j]$ must hold.

©Jesper Larsson Träff

Lemma:
For any i, $0 \leq i < n+m$, there are unique j and k, j+k=i, such that

- Either j=0 or $A[j-1] \leq B[k]$, and
- Either k=0 or $B[k-1] < A[j]$

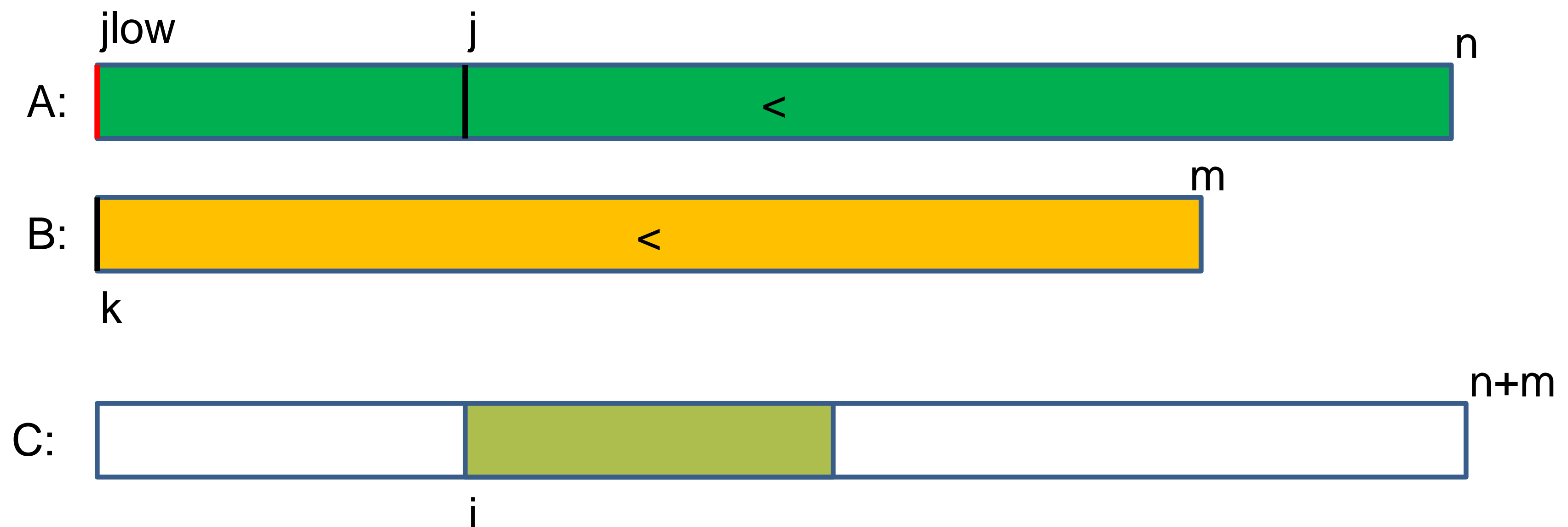
These j and k fulfill merge(A[0,…,j-1],B[0,…,k-1]) = C[0,…,i-1]

The co-ranking algorithm uses the lemma in a binary-search like fashion to find the unique co-ranks of the given i

Christian Siebert, Jesper Larsson Träff: Perfectly Load-Balanced, Stable, Synchronization-Free Parallel Merge. Parallel Processing Letters 24(1) (2014)

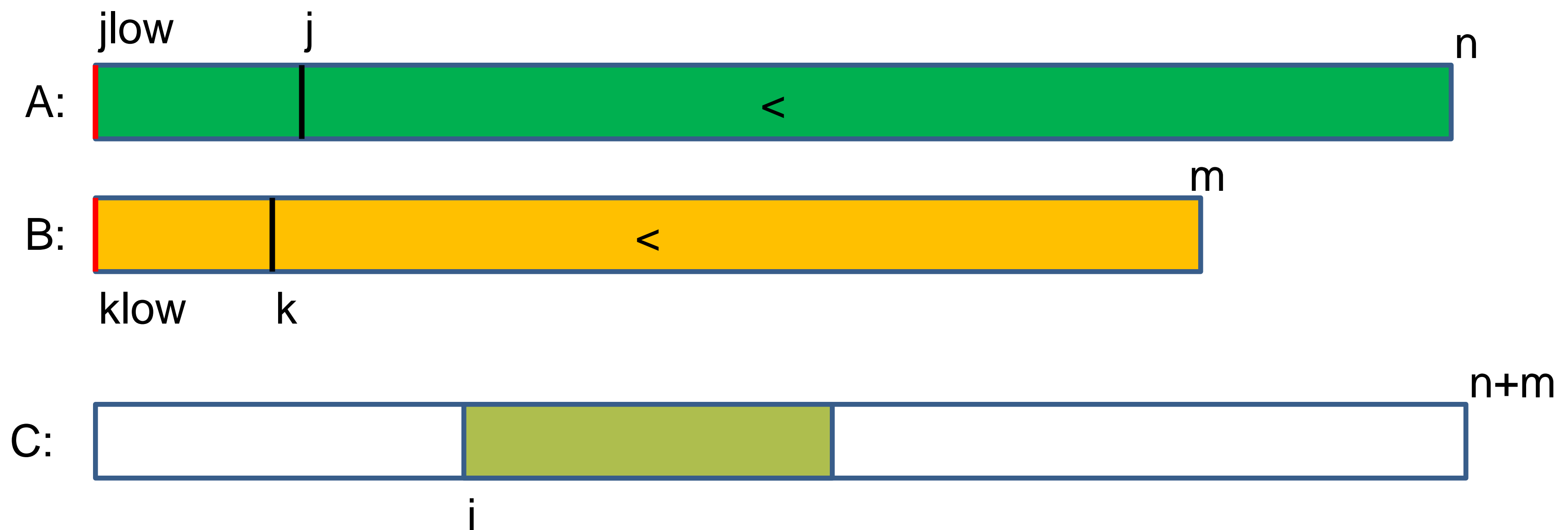
Start by assuming all C elements from A: Set j=min(i,n), by the invariant k=i-j

If A[j-1]>B[k]: j was too large, halve it (need jlow), increase k correspondingly



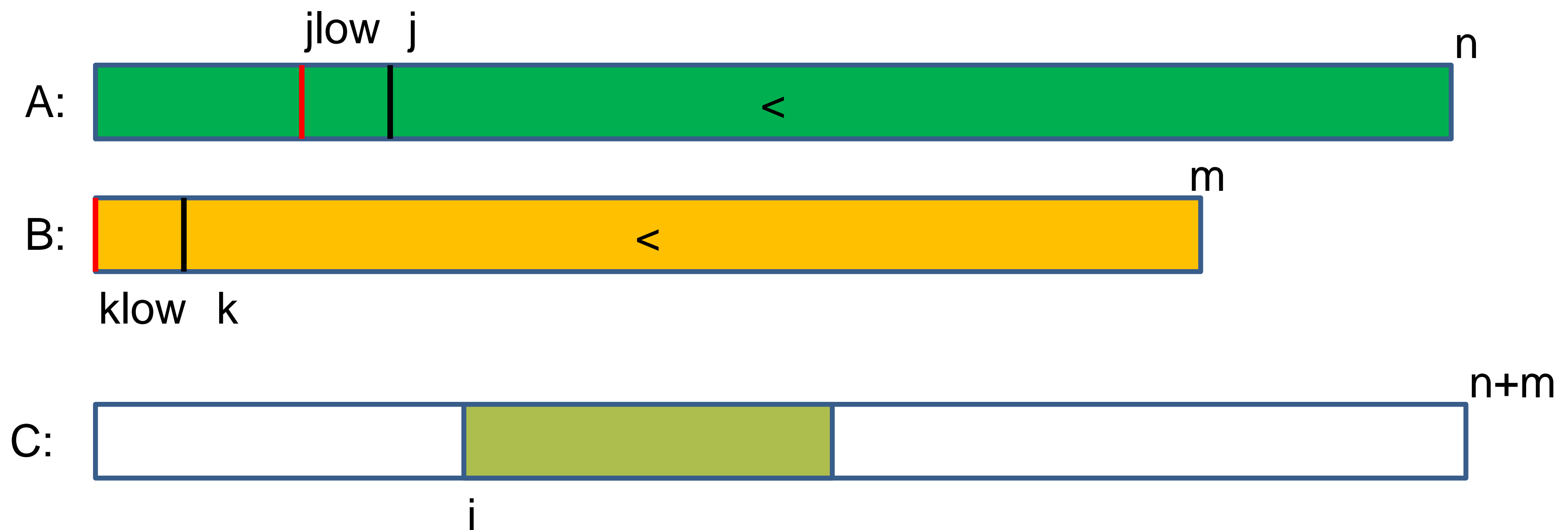
Start by assuming all C elements from A: Set j=min(i,n), by the invariant k=i-j

If B[k-1]≥A[j]: k was too large, halve it (need klow), increase j correspondingly



Start by assuming all C elements from A: Set j=min(i,n), by the invariant k=i-j

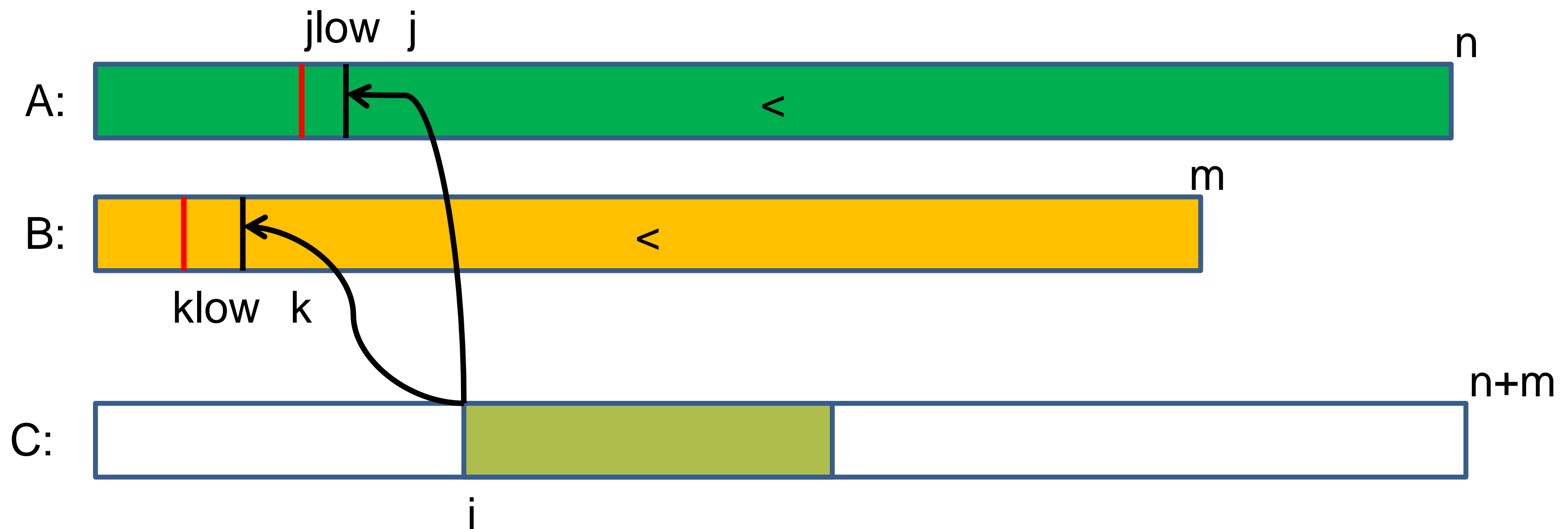
If B[k-1]≥A[j]: k was too large, halve it, increase j correspondingly



When halving j, j=(j-jlow)/2, klow is set to old value of k
When halving k, k=(k-klow)/2, jlow is set to old value of j

Start by assuming all C elements from A: set j=min(i,n), by the invariant k=i-j

Iterate until j and k are found that satisfy the lemma. This happens after at lost log(n+m) iterations



The co-ranking algorithm determines co-ranks j and k for given index i and ordered arrays A and B

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

Merge solution: Issues and observations

- Merging as a (data dependent, adaptive) load balancing problem: divide the two sequences into parts of combined size (n+m)/p that can be merged independently

- Convenient to assume a shared-memory programming model abstraction: merge, binary search, and co-ranking can be given straight-forward, sequential implementations
- Programming model must support allocation of p processing elements to array indices
- Synchronization necessary after certain steps (barrier)

- In a distributed memory programming model, binary search and co-ranking less obvious (communication needed to access parts of input stored with other processes); some data redistribution may be necessary before local merge

Theorem:
On a shared-memory system, two ordered sequences of size n and m can be merged in time $O((n+m)/p+\log n)$

Parallelization (of merge problem):

- Focus on the problem
- Consider potential for parallelization of known sequential algorithm, new idea if necessary
- Make parallel work comparable to sequential work
- Look for good load balance
- Minimize number of synchronization points
- (Communication: not yet seen)
- Use sequential algorithms as subalgorithms

Parallelization (of merge problem):

- Focus on the problem
- Consider potential for parallelization of known sequential algorithm, new idea if necessary
- Make parallel work comparable to sequential work
- Look for good load balance
- Minimize number of synchronization points
- (Communication: not yet seen)
- Use sequential algorithms as subalgorithms

"Automatic parallelization" will most probably not work. The needed preprocessing idea (binary search, co-ranking) is found nowhere in the sequential algorithm

Foster‘s “methodology”

Often cited, “general”, 4-step strategy for parallelizing computations

Ian Foster. Designing and building parallel programs - concepts and tools for parallel software engineering. Addison-Wesley 1995

1. Partitioning: Divide the computation into independent tasks
2. Communication: Determine communication needed between tasks
3. Agglomeration/aggregation: Combine tasks and communications together into larger (independent) chunks
4. Mapping: Assign tasks and communications to processes, threads, …

1. Partitioning: Divide the computation into independent tasks
2. Communication: Determine communication needed between tasks
3. Agglomeration/aggregation: Combine tasks and communications together into larger (independent) chunks
4. Mapping: Assign tasks and communications to processes, threads, …

Rules of thumb, important issues to consider; but unspecific.
Parallelization is problem/algorithm and architecture dependent

There is no general strategy for parallelizing an algorithm, or for finding the right algorithm to parallelize

Foster's "methodology" applied to the merge problem

1. Partitioning: Algorithmic idea needed; rank/co-rank exposes "tasks" (blocks that can be merged) to be performed independently in parallel
2. Communication: How to rank/co-rank? Which data needs to be exchanged before blocks can be merged
3. Agglomeration/aggregation: Ranking/binary search per element too fine-grained, too much communication, too much work. Divide into larger blocks of size n/p, m/p
4. Mapping: Processors close to blocks merge blocks

Four steps only partially applicable to the developed merge algorithm (later: implementation). Helpful?

Oblivious merging: Bitonic merge/Even-odd merge

Problem with merge by co-ranking and rank-based algorithms:

- Many processors may need to read the same array elements at the same time. What if this is not allowed (EREW PRAM)? What if this is not efficient (by serialization when several processors read the same value)?
- What if only O(p) elements need to be merged on p processors? (Solution 1 can be of help here)

(Data) Oblivious parallel algorithms

Definition:
Parallel algorithm is oblivious if its data access pattern (array accesses, communication) is independent of (oblivious to) actual data, dependent only on n and p

Two classical, oblivious merge algorithms

- Even/odd merge
- Bitonic merge

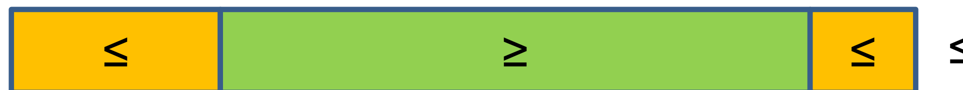
Kenneth E. Batcher: Sorting Networks and Their Applications. AFIPS Spring Joint Computing Conference 1968: 307-314
Kenneth E. Batcher: On Bitonic Sorting Networks. ICPP (1) 1990: 376-379

Bitonic merge

A sequence $(a_0,a_1,\ldots,a_{n-1})$ is bitonic if either

1. There is an index i such that $a_0\leq a_1\leq\ldots\leq a_i$ and $a_{i+1}\geq a_{i+2}\geq\ldots\geq a_{n-1}$
2. There is a cyclic shift of the sequence, such that 1. holds

Case 1

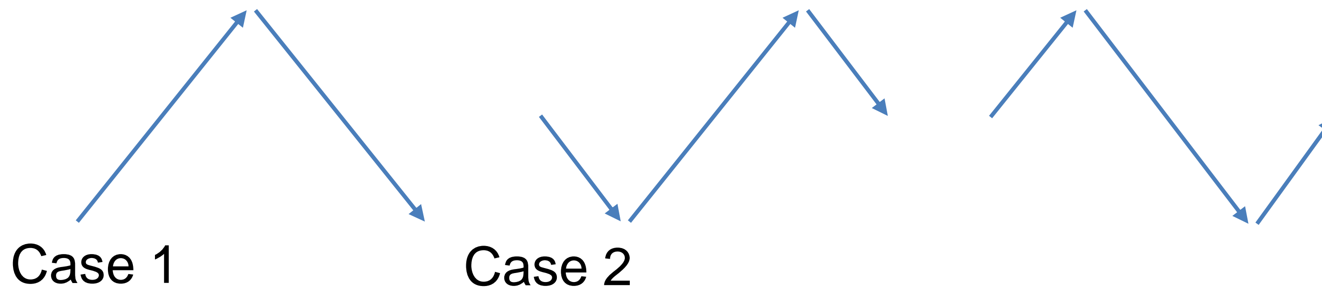


≤

Case 2

A sequence $(a_0,a_1,...,a_{n-1})$ is bitonic if either

1. There is an index i such that $a_0 \leq a_1 \leq ... \leq a_i$ and $a_{i+1} \geq a_{i+2} \geq ... \geq a_{n-1}$
2. There is a cyclic shift of the sequence, such that 1. holds



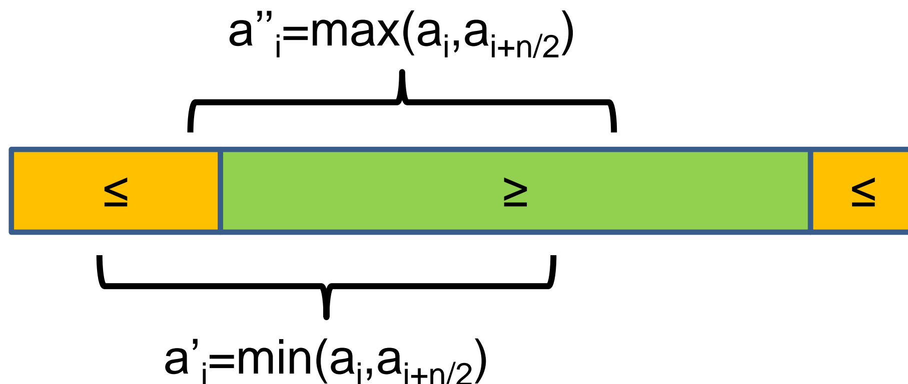
Lemma:

Let $a = (a_0,a_1,\ldots,a_{n-1})$ be a bitonic sequence of even length n. The two sequences

- $a' = (\min(a_0,a_{n/2}),\min(a_1,a_{n/2+1}),\ldots,\min(a_{n/2-1},a_{n-1}))$
- $a'' = (\max(a_0,a_{n/2}), \max(a_1,a_{n/2+1}),\ldots,\max(a_{n/2-1},a_{n-1}))$

of length n/2 are

1. bitonic, and
2. $a' \leq a''$ (all elements of a' no larger than all elements of a'')



Example:

- a = (1,1,2,3,4,7,7,6,5,4,4,3) = (1,1,2,3,4,7)||(7,6,5,4,4,3)
- a' = (1,1,2,3,4,3)
- a'' = (7,6,5,4,4,7)

} Bitonic

Example (sequence cyclically shifted):

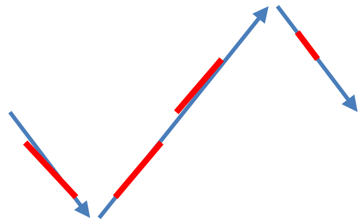
- a = (3,4,7,7,6,5,4,4,3,1,1,2) = (3,4,7,7,6,5)||(4,4,3,1,1,2)
- a' = (3,4,3,1,1,2)
- a'' = (4,4,7,7,6,5)

} Bitonic

Proof of lemma:

1. Any subsequence of a bitonic sequence is bitonic. The subsequences a' and a'' obviously partition a (a' and a'' disjoint, union of a' and a'' is a)
2. Assume there is some $a'_i > a''j$. Exhaustive case analysis in all cases leads to a contradiction

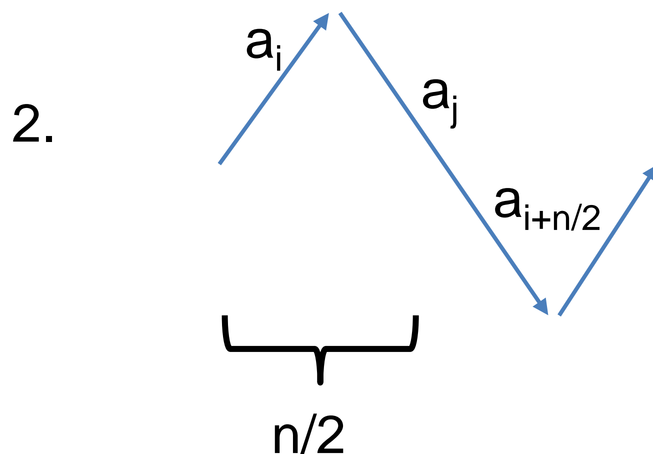
1.



Proof of lemma:

1. Any subsequence of a bitonic sequence is bitonic. The subsequences a' and a'' obviously partitions a (a' and a'' disjoint, union of a' and a'' is a)
2. Assume there is some $a'_i > a''j$. Exhaustive case analysis in all cases leads to a contradiction

2.



Assume $a'_i = \min(a_i, a_{i+n/2}) = a_i$

Then it must be that $a_j < a_i$, contradicting either that a'_i is min, or a''_j is max

Ordering bitonic sequences:

Given bitonic sequence a:

- Split a into sequences a' of minima and sequence a'' of maxima
- Recursively order the two bitonic sequences a' and a''

Lemma can easily be extended to odd length sequences. Virtually repeat the last element a_{n-1} to get an even length sequence (helping observation: Element in a bitonic sequence can be repeated, and still give a bitonic sequence). The bitonic split will put the real a_{n-1} in either a' or a shifted a''.

But note: This extension is no longer oblivious. There are other ways of extending bitonic ordering to odd sequence lengths

Wolfgang J. Paul: A Note on Bitonic Sorting. Inf. Process. Lett. 49(5): 223-225 (1994)

```
bitonic_merge(int a[], int n)
{
  if (n==1) return;

  int nn = n/2; int s = n/2;
  if (n%2==1) { // n odd
    nn++;
    if (a[n/2]<a[n-1]) s++;
  }
  for (i=0; i<n/2; i++) {
    int mina, maxa;
    mina = min(a[i],a[i+nn]);
    maxa = max(a[i],a[i+nn]);
    a[i] = mina; a[i+nn] = maxa;
  }
  bitonic_merge(a,s);
  bitonic_merge(a+s,n-s);
}
```

Let W(n) be the work (number of operations) for the bitonic merge of a sequence a of length n. We have

- $W(1) = O(1)$
- $W(n) = O(n)+2W(n/2)$

with solution $W(n) = O(n \log n)$

Induction hypothesis

Proof:
Expand recursion a few times, and guess solution. By induction, ignoring O-constants:
$W(n) = n + 2(n/2 \log_2 n/2) = n + n \log_2 n/2 = n + n \log_2 n - n = n \log_2 n$

By same arguments, recursion depth is $\mathrm{ceil}(\log_2 n)$

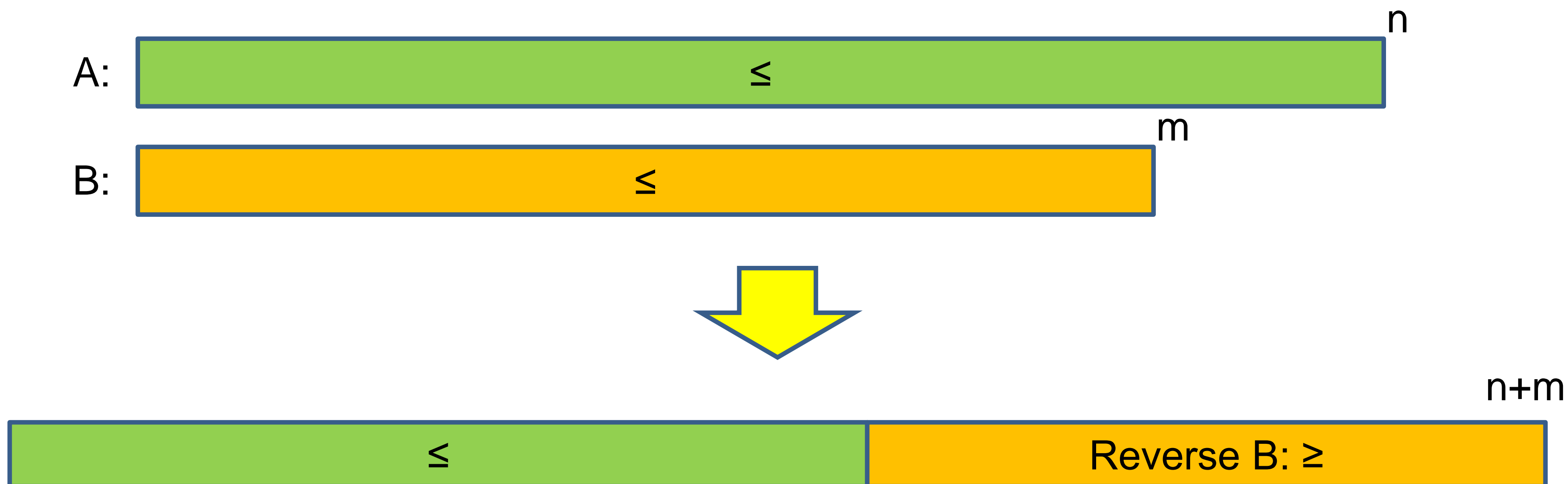
Theorem:
On a shared-memory system, a bitonic sequence of length p can be ordered into a sequence in increasing order in ceil($\log_2 p$) parallel steps and O(p log p) operations. A bitonic sequence of length n can be ordered in time O((n log n)/p+log p).

Bitonic ordering can be done in-place

Note: Bitonic ordering is not stable

When n>p, do bitonic merge recursion until $n/2^k = n/p$, then merge bitonic subsequences of length n/p sequentially on the p processors. Still $2^k = p \Leftrightarrow k = \log_2 p$ and work per recursion step O(n), so total work is O(n log p). The algorithm is not work-optimal

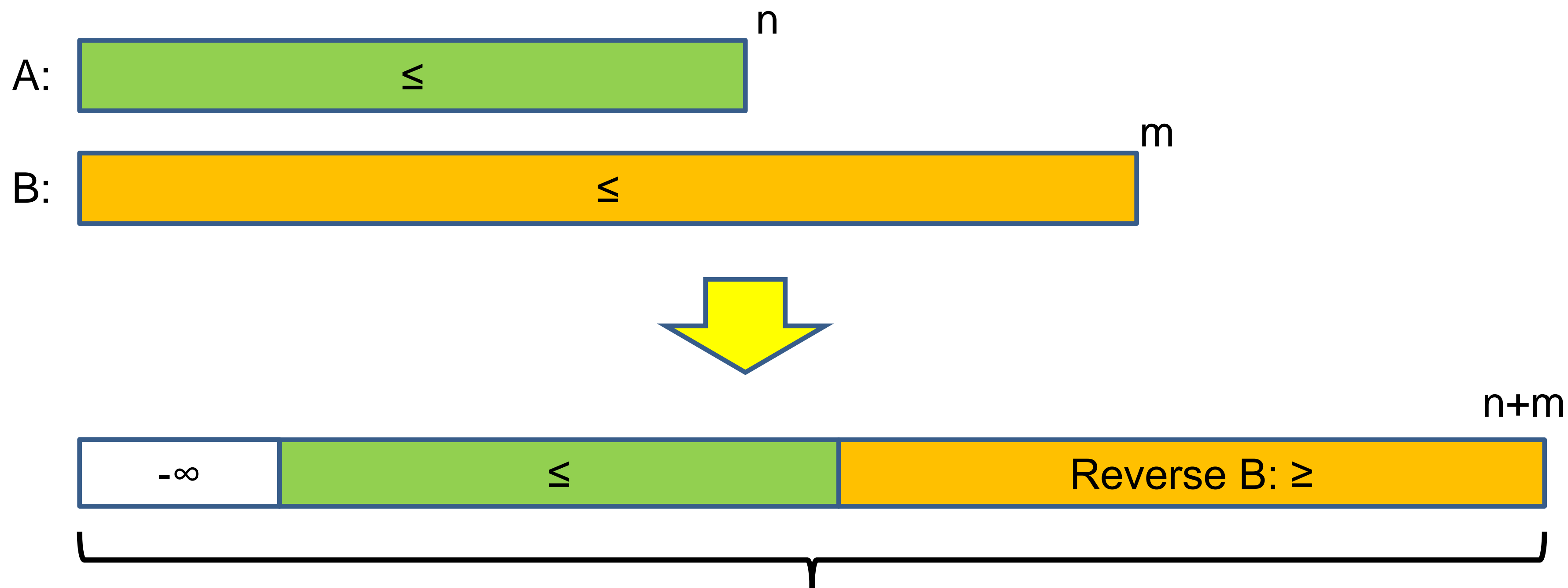
Merging ordered sequences A and B by bitonic ordering



Resulting sequence is bitonic, but n+m may not be a power of two.

To get a bitonic sequence of length a power of two, pad from below with virtual -∞ elements to nearest power of two.

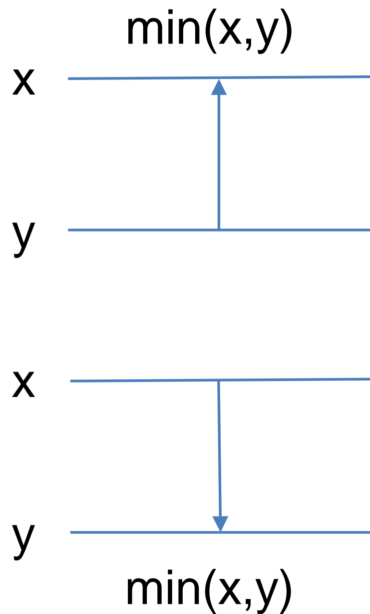
Merging ordered sequences A and B by oblivious bitonic ordering



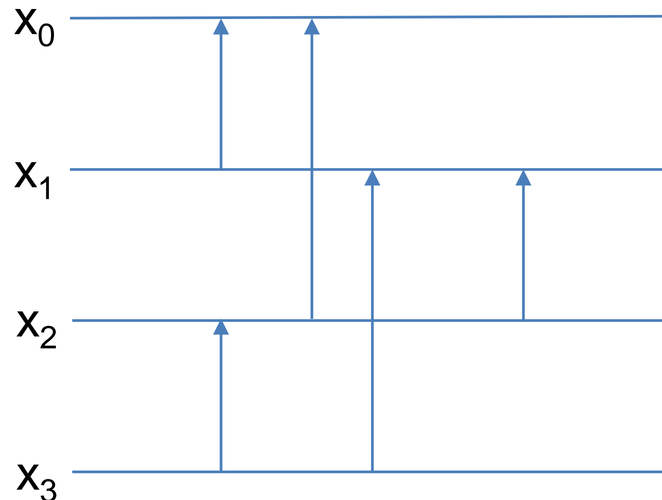
Observation: The virtual elements will stay in front after bitonic split, and can be kept as virtual elements

Comparator networks as a model for parallel sorting

Comparator (min/max)



Sorting network for n=4



Size: Number of comparators (≈ number of operations)
Depth: Longest path from an input to an output

Question:
What is the minimal depth and size required for sorting n number with a sorting network?
Is there a sorting network of depth O(log n) of size O(n log n)?

Long standing, open question in parallel computing

D. E. Knuth: The Art of Computer Programming, Vol. 3. Addison-Wesley, 1973.

Section 5.3.4, Exercise 51 [M50]: Prove that the asymptotic value of $\hat{S}(n)$ is not O(n log n)

Bather's bitonic sorting network has depth $O(\log^2 n)$ and size $O(n \log^2 n)$

Question resolved in 1983 with the O(n log n) size AKS network

Miklós Ajtai, János Komlós, Endre Szemerédi: An O(n log n) Sorting Network. STOC 1983: 1-9
Miklós Ajtai, János Komlós, Endre Szemerédi: Sorting in c log n parallel sets. Combinatorica 3(1): 1-19 (1983)

Complex construction (expander graphs), excessive constants; not practical.
Another milestone in parallel sorting: O((n log n)/p + log n) EREW PRAM mergesort; not practical

Richard Cole: Parallel Merge Sort. SIAM J. Comput. 17(4): 770-785 (1988)
Richard Cole: Correction: Parallel Merge Sort. SIAM J. Comput. 22(6): 1349 (1993)

Reduction and prefix sums in parallel

Reduction problem: Given sequence $x_0, x_1, x_2, \ldots, x_{n-1}$, compute

$$y = \sum_{0\leq i<n} x_i = x_0+x_1+x_2+\ldots+x_{n-1}$$

- x_i integers, real numbers, vectors, structured values…
- "+" any applicable operator, sum, product, min, max, bitwise and, logical and, vector sum, …

Algebraic properties of "+": Associative $[x+(y+z)=(x+y)+z]$, possibly commutative, …

Parallel reduction problem:
Given sequence (array) of elements (x_i), associative operation "+", compute the sum $y = \sum x_i$

Reduction is a fundamental, primitive operation, used in many, many applications (recall: Map-Reduce). Available in some form in most parallel programming models/interfaces as "collective operation"

Collective operation pattern:
Set of processors (threads, processes, …) "collectively" invoke some operation, each contribute a subset of the n elements, process order determine element order

- Reduction-to-one: All processors participate in the operations, resulting "sum" stored with one specific processor ("root")
- Reduction-to-all: All processors participate, results available to all processes
- Reduction-with-scatter: Reduction of vectors, result vector stored in blocks over the processors according to some rule

Definitions:
i'th prefix sum: Sum of the first i elements of x_i sequence

$$y_i = \sum_{0\leq j<i} x_j = x_0+x_1+x_2+\ldots+x_{i-1}$$

a) Exclusive prefix (i>0) sum: up to, but not including x_i in sum (special definition for i=0)

$$y_i = \sum_{0\leq j\leq i} x_j = x_0+x_1+x_2+\ldots+x_i$$

b) Inclusive prefix sum: up to and including x_i in sum.
Note: Inclusive prefix trivially computable from exclusive prefix (add xi), not vice versa unless "+" has inverse

Parallel prefix sums problem: Given sequence x_i, compute all n prefix sums $y_0, y_1, \ldots, y_{n-1}$

Prefix-sums is a fundamental, primitive operation, used for bookkeeping and load balancing purposes (and others) in many, many applications. Available in some form in most parallel programming models/interfaces.

The collective prefix-sums operation often referred to as Scan:
Process i, $i \le i < p$, has x_i

- Scan: Process i computes inclusive prefix sum y_i
- Exscan: Process i computes exclusive prefix sum y_i

Reduction, Scan:
Input sequence $x_0, x_1, x_2, \ldots, x_{n-1}$ in array, distributed array, … in form suitable to programming model

©Jesper Larsson Träff

Parallel reduction problem:
Given sequence (array) of elements, associative operation "+", compute the sum $y = \sum x_i$

Parallel prefix-sums problem:
Compute all n (inclusive or exclusive) prefix sums $y_0, y_1, \ldots, y_{n-1}$

Sequentially, both problems can be solved in O(n) operations, and n-1 applications of "+".

This is optimal (best possible), since the output depends on each input x_i. Complexity is $\Theta(n)$.

How to solve prefix sums problem efficiently in parallel?

- Total number of operations (work) proportional to $Tseq(n) = O(n)$
- Total number of actual "+" applications close to n-1
- Parallel time $Tpar(n) = O(n/p+T\infty(n))$ for large range of p
- As fast as possible, $T\infty(n) = O(\log n)$

Remark:
In most reasonable architecture models, $\Omega(\log n)$ would be a lower bound on the parallel running time for a work-optimal solution

Sequential solution (both reduction and scan): Simple scan through array with running sum

```
for (i=1; i<n; i++) {
  x[i] = x[i-1]+x[i];
}
```

Direct solution, not "best sequential implementation": 2(n-1) memory reads

Tseq(n) = n-1 summations, O(n), 1 read, 1 write per iteration

```
register int sum = x[0];
for (i=1; i<n; i++) {
  sum += x[i]; x[i] = sum;
}
```

Register solution possibly better, but far from best

Questions: What can the compiler do? How much dependence on basetype (int, long, float, double)? On content?

Some results (the two solutions and the compiler):

Implementation with OpenMP:

```
traff 60> gcc -o pref -fopenmp <optimization> …

traff 61> gcc --version
gcc (Debian 4.7.2-5) 4.7.2
Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying
conditions.  There is NO warranty; not even for
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Execution on small Intel system

```
traff 62> more /proc/cpuinfo
…
Model name:  Intel (R) Core (TM) i7-2600 CPU @ 3.40 GHz
```

```
traff 98> ./pref -n 100000 -t 1
n is 100000 (0 MBytes), threads 1(1)
Basetype 4, block 1 MBytes, block iterations 0
Algorithm Seq straight Time spent 502.37 microseconds
(min 484.67 max 566.68)
Algorithm Seq Time spent 379.09 microseconds (min 296.97
max 397.58)
Algorithm Reduce Time spent 249.39 microseconds (min
210.75 max 309.59)
traff 99> ./pref -n 1000000 -t 1
n is 1000000 (3 MBytes), threads 1(1)
Basetype 4, block 1 MBytes, block iterations 3
Algorithm Seq straight Time spent 3532.19 microseconds
(min 2875.97 max 4552.18)
Algorithm Seq Time spent 2304.72 microseconds (min
2256.46 max 2652.57)
Algorithm Reduce Time spent 1625.14 microseconds (min
1613.68 max 1645.12)
```

Small custom benchmark, omp_get_wtime() to measure time, 25 repetitions, average running times of prefix-sums function (including function invocation)

Time in microseconds

		int		
	No opt		-O3	
	Direct	Register	Direct	Register
100,000	502	379	73	72
1,000,000	3532	2304	615	552
10,000,000	28152	23404	5563	5499

Optimizer (-O3) can do a lot (discover register/running sum improvement)

Lessons:
For "best sequential implementation", explore what compiler can do (a lot). Document used compiler options (for reproducibility)

Time in microseconds

		double		
	No opt		-O3	
	Direct	Register	Direct	Register
100,000	451	653	145	144
1,000,000	3926	4454	1466	1162
10,000,000	30411	44344	10231	10001

Surprisingly, non-optimized, direct solution faster than register running sum. Optimization a must: Look into other optimization possibilities (flags)

Reduction application: Cutoff computation

```
// Parallelizable part
do {
   for (i=0; i<n; i++) {
        x[i] = f(i);
   }
   // convergence check
   done = …;
} while (!done)
```

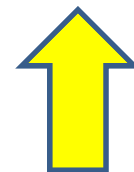
done: if x[i]<ε for all i

Each process locally computes

localdone = (x[i]<ε) for all local i

done = allreduce(localdone,AND);

Local input



Associative reduction operation

Collective operation:
perform reduction over
set of involved processes,
distribute result to all
processes

Prefix-sums application: Array compaction, load balancing

Given arrays `a` and `active`, execute loop efficiently in parallel:

```
for (i=0; i<n; i++) {
  if (active[i]) a[i] = f(b[i]+c[i]);
}
```

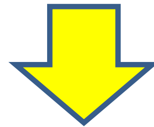
Work O(n) for the loop plus O(|`active`| `f`) for the function evaluations, where |`active`| is the number of indices for which `active[i]` is true

Given arrays `a` and `active`, execute loop efficiently in parallel:

```
for (i=0; i<n; i++) {
  if (active[i]) a[i] = f(b[i]+c[i]);
}
```

Data parallel computation

No dependencies between loop iterations



Processor j, 0≤j<p

```
for (i=j*(n/p); i<(j+1)*(n/p); i++) {
  if (active[i]) a[i] = f(b[i]+c[i]);
}
```

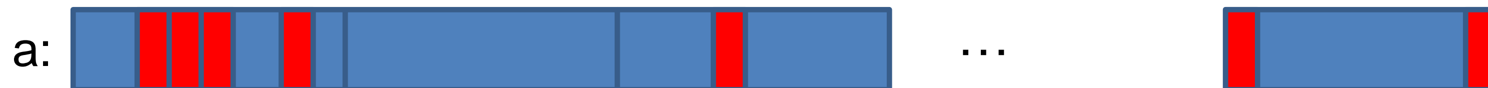
Static assignment of work to processors

Work: O(n+|active| f). Time Tpar(p,n)?

Problem with static division of loop iteration space into fixed blocks

Processor j, 0≤j<p

```
for (i=j*(n/p); i<(j+1)*(n/p); i++) {
  if (active[i]) a[i] = f(b[i]+c[i]);
}
```

a: 

Loop split across p processors can be inefficient: Some processors (those with `active[i]` false) do little work, while others take the major part. In worst case, Tpar(p,n) = O(|`active`| `f`), if all the work is done by one processor. Typical load balancing problem

Given arrays `a` and `active`, execute loop efficiently in parallel:

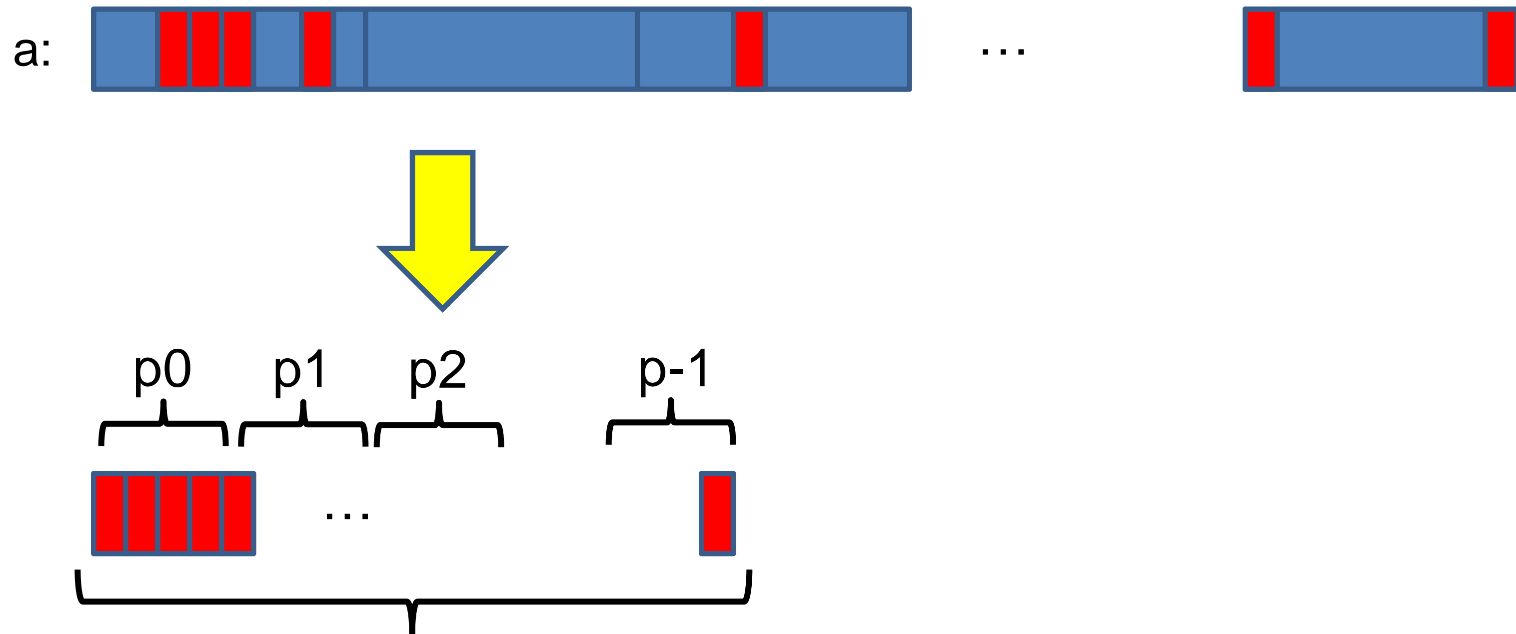
```
for (i=0; i<n; i++) {
  if (active[i]) a[i] = f(b[i]+c[i]);
}
```

Work O(n) for the loop plus O(|`active`| `f`) for the function evaluations, where |`active`| is the number of indices for which `active[i]` is true

Solution:
Reduce work to O(|`active`| `f`) by compacting active indices into consecutive positions of new array, parallelize over smaller array

Iteration space:

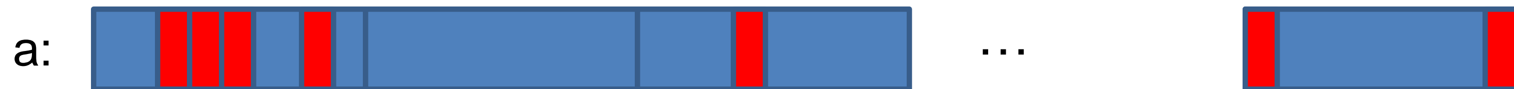
```
for (i=0; i<n; i++) if (active[i]) a[i] = f(b[i]+c[i]);
```



Smaller array of active indices: split into evenly sized blocks of size |`active`|/p

Iteration space:

```
for (i=0; i<n; i++) if (active[i]) a[i] = f(b[i]+c[i]);
```



Approach: Count and index active elements

1. Mark active elements with index 1
2. Mark non-active element with index 0
3. Exclusive prefix-sums operation over new index array
4. Store original indices in smaller array

Work, time:

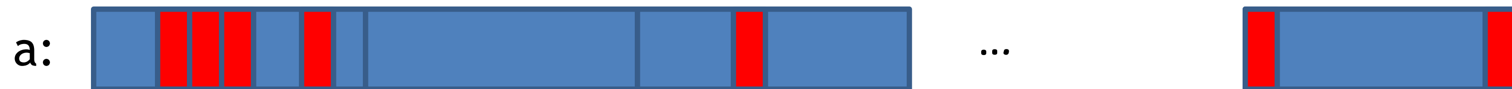
1+2.O(n), O(n/p)

3. We don't know yet
4. O(n), O(n/p)

Last prefix sum is (almost) equal to `|active|`

Iteration space:

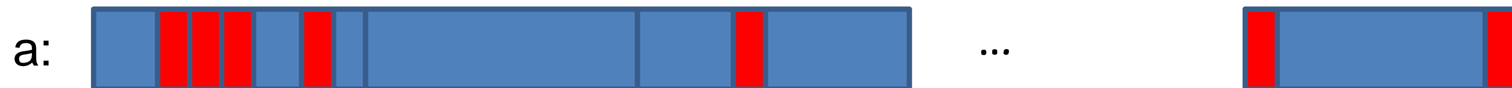
```
for (i=0; i<n; i++) if (active[i]) a[i] = f(b[i]+c[i]);
```




```
for (i=0; i<n; i++) index[i] = active[i] ? 1 : 0;
Exscan(index,n); // exclusive prefix computation
m = index[n-1]+(active[n-1) ? 1 : 0);
for (i=0; i<n; i++) {
  if (active[i]) oldindex[index[i]] = i;
}
```

Iteration space:

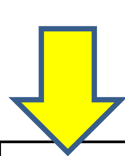
```
for (i=0; i<n; i++) if (active[i]) a[i] = f(b[i]+c[i]);
```




```
for (i=0; i<n; i++) index[i] = active[i] ? 1 : 0;
Exscan(index,n); // exclusive prefix computation
m = index[n-1]+(active[n-1) ? 1 : 0);
for (i=0; i<n; i++) {
  if (active[i]) oldindex[index[i]] = i;
}
```

```
for (j=0; j<m; j++) {
  i = oldindex[j];
  a[i] = f(b[i]+c[i]);
}
```

1. First load balance (prefix-sums)
2. Then execute (data parallel computation)



Data parallel computation over independent indices; concrete assignment to processors ignored here

```
par (i=0; i<n; i++) index[i] = active[i] ? 1 : 0;
Exscan(index,n); // exclusive prefix computation
m = index[n-1]+(active[n-1) ? 1 : 0);
par (i=0; i<n; i++) {
  if (active[i]) oldindex[index[i]] = i;
}
```

```
par (j=0; j<m; j++) {
  i = oldindex[j];
  a[i] = f(b[i]+c[i]);
}
```

1. First load balance (prefix-sums)
2. Then execute (data parallel computation)

Work: O(n) + O(|`active`| `f`)
Time: O(n/p) + Texscan(p,n) + O((|`active`| `f`)/p)

Prefix-sums application: Partitioning for Quicksort

Task parallel Quicksort algorithm

Quicksort(a,n):
1. Select pivot a[k]
2. Partition a into a[0,…,n1-1], a[n1,…,n2-1], a[n2,…,n-1] of elements smaller, equal, and larger than pivot
3. In parallel: Quicksort(a,n1), Quicksort(a+n2,n-n2)

Running time T_∞ (assuming good choice of pivot, at most n/2 elements in either segment):

$$T_\infty(n) \leq T_\infty(n/2)+O(n) \text{ with solution } T_\infty(n) = O(n)$$

Maximum speedup over sequential O(n log n) Quicksort is therefore O(log n). Need to parallelize partition step

Partition:

1. Mark elements smaller than a[k], compact into a[0,…,n1-1]
2. Mark elements equal to a[k], compact into a[n1,…,n2-1]
3. Mark elements greater than a[k], compact into a[n2,…,n-1]

```
par (i=0; <n; i++) index[i] = (a[i]<a[k]) ? 1 : 0;
Exscan(index,n); // exclusive prefix computation
n1 = index[n-1]+(active[n-1] ? 1 : 0);
par (i=0; i<n; i++) {
  if (a[i]<a[k]) aa[index[i]] = a[i];
}
// same for equal to and larger than pivot elements
…
// copy back
par (i=0; i<n; i++) a[i] = aa[i];
```

Partition:
Three applications of prefix-sums (Exscan). How fast can prefix-sums be computed?

Answer (how? See later):
In time O(n/p+log n) operations with p processors for array of n elements

Remaining load balancing problem:
Assign processors proportionally to smaller and larger segments, Quicksort-recurse in parallel

OpenMP tasks (or Cilk) will help

Task parallel Quicksort algorithm with parallel partition

Quicksort(a,n):
1. Select pivot a[k]
2. Parallel Partition of a into a[0,…,n1-1], a[n1,…,n2-1], a[n2,…,n-1] of elements smaller, equal, and larger than pivot
3. In parallel: Quicksort(a,n1), Quicksort(a+n2,n-n2)

$T_\infty(n) \leq T_\infty(n/2) + O(\log n)$ with solution $T_\infty(n) = O(\log^2 n)$

Maximum possible speed-up is now $O(n/\log n)$

Prefix-sums application: Load balancing for merge algorithm

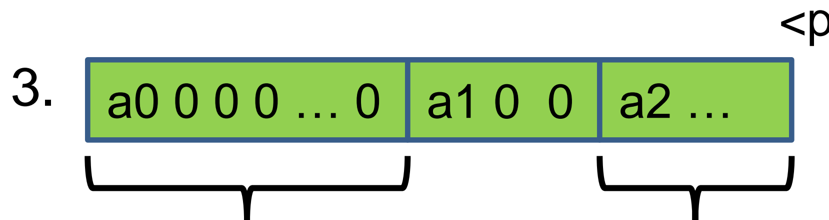
Bad segments: rank[i+1]-rank[i]>m/p

Possible solution:

1. Compute total size of bad segments (parallel reduction)
2. Assign a proportional number of processors to each bad segment
3. Compute array of size O(p), each entry corresponding to a processor assigned to a bad segment: start index of segment, size of segment, relative index in segment

Processors corresponding to bad indices

Prefix-sums compaction, Reduction, bad segment size bi



Processors allocated to bi

Number of processors for bi is ai ≈ p*bi/b

4.

A0 A0 … A0	A1…A1	A2 …
0 1 … 2	3 4 …	7 8 …
ix0 ix0 …	ix1 ix1	ix2…
Start, size		

1. $Ai = \sum_{0\leq j<i} a_j$ (exclusive prefix-sums)
2. Running index by prefix-sums (relative index: running ix-A(i-1))
3. Start index by prefix-sums with max-operation
4. Bad segment start and size, prefix-sums

Three theoretical solutions to the parallel prefix-sums problem

1. Recursive: Fast, work-optimal
2. Iterative: Fast, work-optimal
3. Doubling: Fast(er), not work-optimal (but still useful)

Questions:

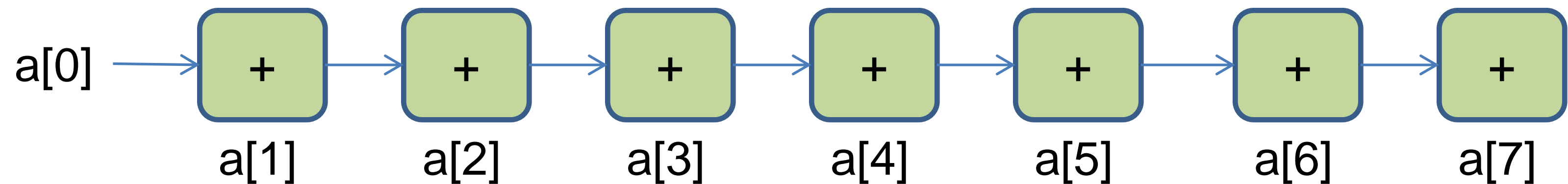
- How fast can these algorithms really solve the prefix-sums problem?
- How many operations do they require (work)?

All three solutions quite different from sequential solution

Key to solution: Associativity of “+”:

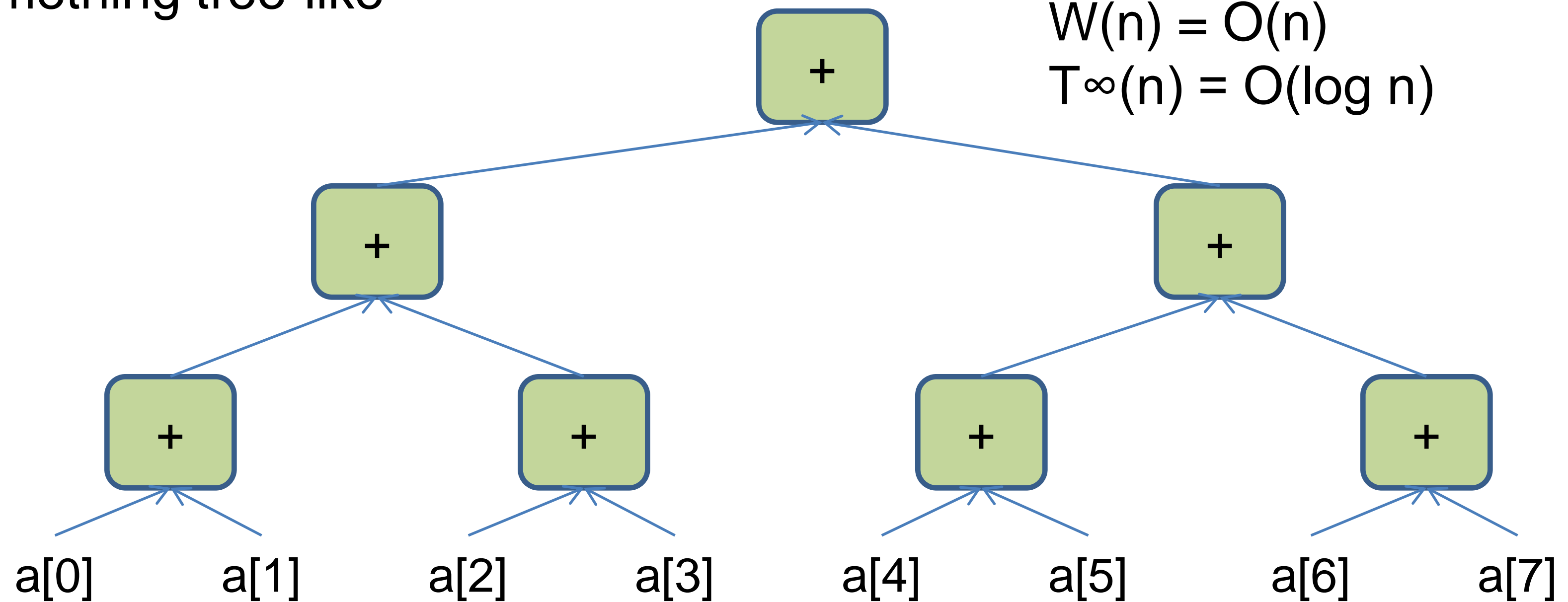
$x_0+x_1+x_2+\ldots+x_{n-2}+x_{n-1} = ((x_0+x_1)+(x_2+\ldots))+\ldots+(x_{n-2}+x_{n-1}))$

Instead of $W(n) = O(n)$, $T\infty(n) = O(n)$



something tree-like

$W(n) = O(n)$
$T\infty(n) = O(\log n)$



©Jesper Larsson Träff

1. Recursive solution: Sum pairwise, recurse on smaller problem

```
Scan(x,n)
{
  if (n==1) return;

  for (i=0; i<n/2; i++) y[i] = x[2*i]+x[2*i+1];

  Scan(y,n/2);

  x[1] = y[0];
  for (i=1; i<n/2; i++) {
    x[2*i] = y[i-1]+x[2*i];
    x[2*i+1] = y[i];
  }
  if (odd(n)) x[n-1] = y[n/2-1]+x[n-1];
}
```

Reduce problem

Solve recursively

Take back

1. Recursive solution: Parallelization

```
Scan(x,n)
{
  if (n==1) return;

  for (i=0; i<n/2; i++) y[i] = x[2*i]+x[2*i+1];

  Scan(y,n/2);

  x[1] = y[0];
  for (i=1; i<n/2; i++) {
    x[2*i] = y[i-1]+x[2*i];
    x[2*i+1] = y[i];
  }
  if (odd(n)) x[n-1] = y[n/2-1]+x[n-1];
}
```

Data parallel loop

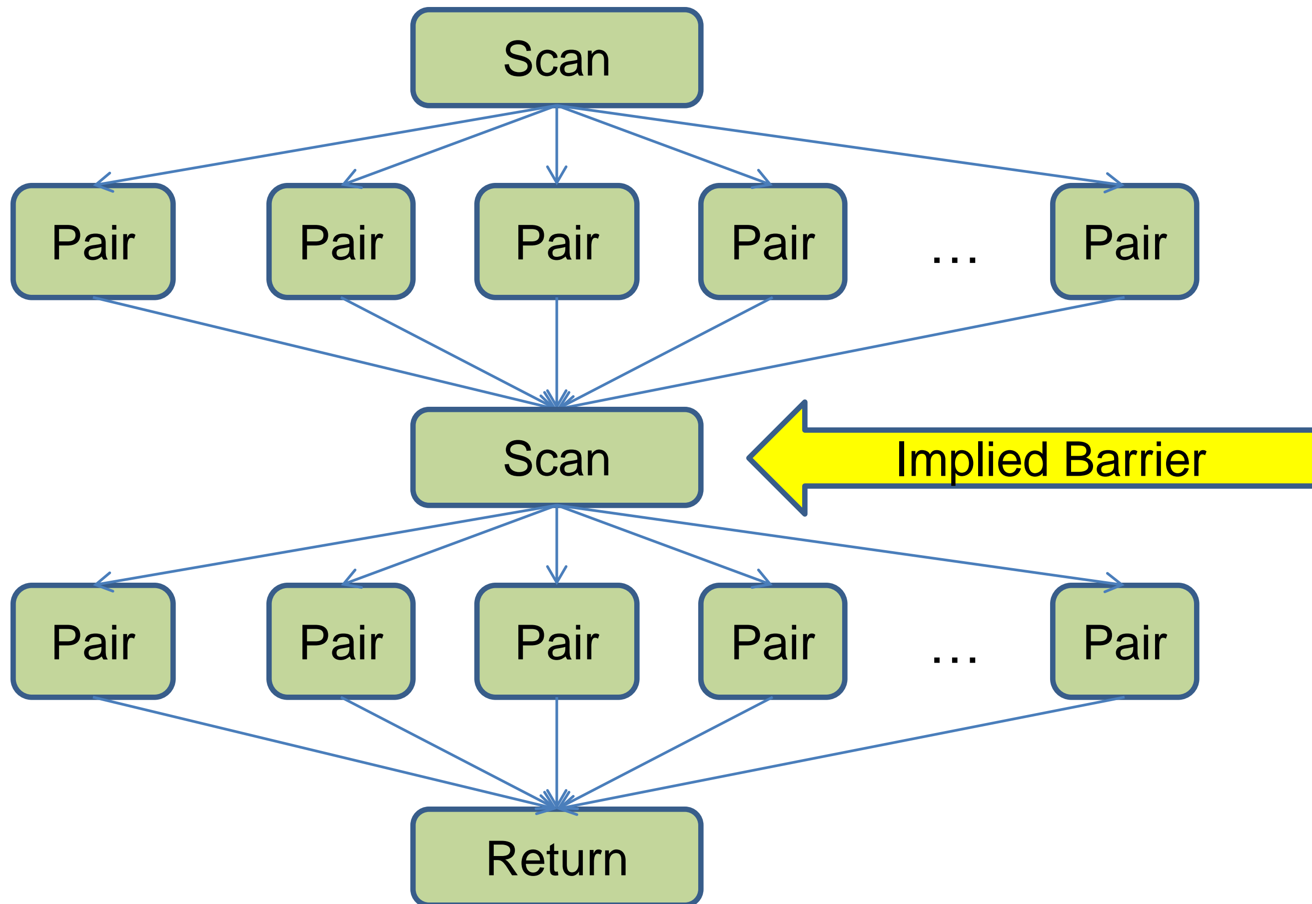
All processors must have completed loop before call

Implicit or explicit "barrier"

Data parallel loop

All processors must have completed call before loop

Fork-join parallelism, parallel recursive calls



1. Recursive solution: Complexity and correctness

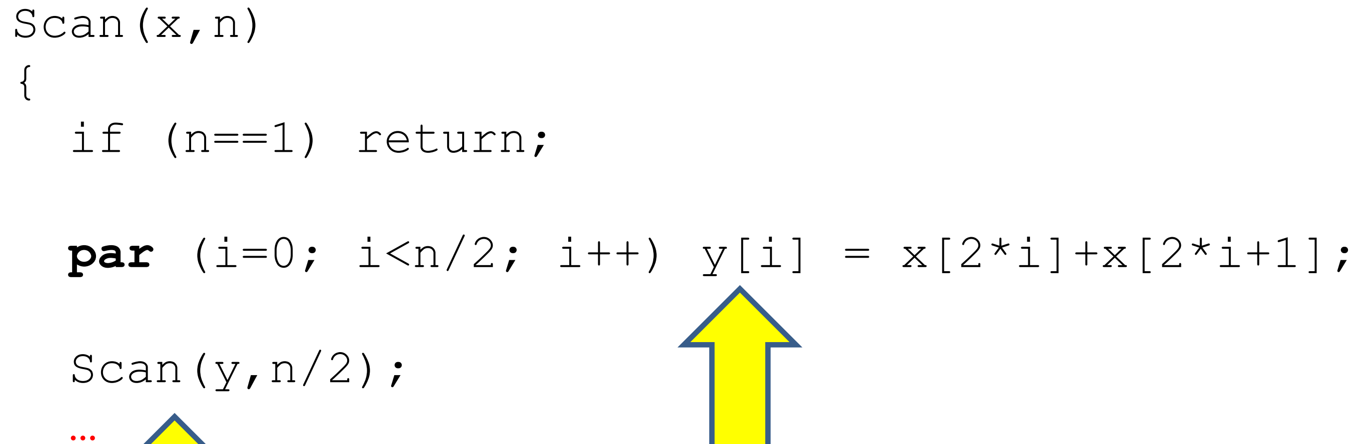
```
Scan(x,n)
{
  if (n==1) return;

  for (i=0; i<n/2; i++) y[i] = x[2*i]+x[2*i+1];

  Scan(y,n/2);
  ...
```

Solve same type of problem, now of size n/2

Total number of “+” operations W(n) satisfies:

- $W(1) = O(1)$
- $W(n) \leq n+W(n/2)$

1. Recursive solution: Complexity and correctness



O(n) operations, perfectly parallelizable: O(n/p)

Solve same type of problem, now of size n/2

Total number of "+" operations W(n) satisfies:

- $W(1) = O(1)$
- $W(n) \leq n+W(n/2)$

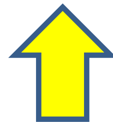
Total number of operations:

- $W(1) = O(1)$
- $W(n) \leq n+W(n/2)$

Expand recurrence: $W(n) = n+W(n/2) = n + (n/2) + (n/4)+\ldots+1$

Guess solution: $W(n) \leq 2n$ (recall: geometric series…)

Verify guess by induction:
$W(1) = 1 \leq 2$
$W(n) = n+W(n/2) \leq n+2(n/2) = n+n = 2n$



By induction hypothesis

1. Recursive solution: Complexity and correctness

```
Scan(x,n)
{
  if (n==1) return;

  par (i=0; i<n/2; i++) y[i] = x[2*i]+x[2*i+1];

  Scan(y,n/2);
  ...
```

O(1) time steps, if n/2 processors available

Solve same problem of size n/2

Number of recursive calls T(n) (which will be $T\infty$) satisfies

- $T(1) = O(1)$
- $T(n) \leq 1+T(n/2)$

Number of recursive calls

- $T(1) = O(1)$
- $T(n) \leq 1+T(n/2)$

$T(n) = 1+T(n/2) = 1+1+T(n/4) = 1+1+1+T(n/8) = \ldots$

$2^k \geq n \Leftrightarrow k \geq \log_2 n$

Guess: $T(n) \leq 1+\log_2 n$

Guess is correct, by induction:

$T(1) = 1 = 1+\log_2(1) = 1+0$

$T(n) = 1+T(n/2) \leq 1+(1+\log_2(n/2)) = 1+(1+\log_2 n - \log_2(2)) =$

$1+(1+\log_2 n - 1) = 1+\log_2 n$

```
Scan(x,n)
{
  if (n==1) return; // base
  par (i=0; i<n/2; i++) y[i] = x[2*i]+x[2*i+1];
  Scan(y,n/2);
  x[1] = y[0];
  par (i=1; i<n/2; i++) {
    x[2*i] = y[i-1]+x[2*i];
    x[2*i+1] = y[i];
  }
  if (odd(n)) x[n-1] = y[n/2-1]+x[n-1];
}
```

Claim:
The algorithm computes the inclusive prefix-sums of $x_0,x_1,x_2,\ldots$, that is, $x_i = \Sigma_{0\leq j\leq i}X_j$, where X_j is the value of x_j before the call

```
Scan(x,n)
{
  if (n==1) return; // base
  par (i=0; i<n/2; i++) y[i] = x[2*i]+x[2*i+1];
  Scan(y,n/2);
  x[1] = y[0];
  par (i=1; i<n/2; i++) {
    x[2*i] = y[i-1]+x[2*i];
    x[2*i+1] = y[i];
  }
  if (odd(n)) x[n-1] = y[n/2-1]+x[n-1];
}
```

Proof by induction:
Base n=1 is correct, algorithm does nothing

```
Scan(x,n)
{
  if (n==1) return; // base
  par (i=0; i<n/2; i++) y[i] = x[2*i]+x[2*i+1];
  Scan(y,n/2); // by induction hypothesis
  x[1] = y[0];
  par (i=1; i<n/2; i++) {
    x[2*i] = y[i-1]+x[2*i];
    x[2*i+1] = y[i];
  }
  if (odd(n)) x[n-1] = y[n/2-1]+x[n-1];
}
```

Proof by induction:
By induction hypothesis, $y_i = \sum_{0\leq j\leq i} Y_j$, where Y_j is the value of y_j before the recursive Scan call, so $y_i = \sum_{0\leq j\leq i} Y_j = \sum_{0\leq j\leq i}(X_{2j}+X_{2j+1})$

```
Scan(x,n)
{
  if (n==1) return; // base
  par (i=0; i<n/2; i++) y[i] = x[2*i]+x[2*i+1];
  Scan(y,n/2); // by induction hypothesis
  x[1] = y[0];
  par (i=1; i<n/2; i++) {
    x[2*i] = y[i-1]+x[2*i];
    x[2*i+1] = y[i];
  }
  if (odd(n)) x[n-1] = y[n/2-1]+x[n-1];
}
```

By induction hypothesis $y_i = \sum_{0\leq j\leq i} Y_j = \sum_{0\leq j\leq i}(X_{2j}+X_{2j+1})$

Thus, $x_i = y_{(i-1)/2}$ for i odd, and $x_i = y_{i/2-1}+X_i$ for i even fulfill the claim. This is what the algorithm computes after the recursive call

1. Recursive solution: Summary

- With enough processors, $T\infty(n) = 2 \log n$ parallel steps (recursive calls) needed, two barrier synchronizations per recursive call
- Number of operations, $W(n) = O(n)$, all perfectly parallelizable (data parallel)

- $Tpar(p,n) = W(n)/p + T\infty(n) = O(n/p+\log n)$

- Linear speed-up up to $Tseq(n)/T\infty(n) = n/\log n$ processors

1. Recursive solution: Summary (practical considerations)

Advantages:

- Smaller y array may fit in cache, pair-wise summing has good spatial locality (see later)

Drawbacks:

- Space: extra n/2 sized array in each recursive call, n in total
- About 2n “+” operations (compared to sequential scan: n-1)
- 2($\log_2 n$) parallel steps

Aside: Master Theorem for simple, regular recurrence relations

Given recurrence of the form

$$T(n) = a\,T(n/b) + O(n^d \log^e n)$$

for constants $a \geq 1$, $b > 1$, $d \geq 0$, $e \geq 0$, and $T(1)$ some constant. This has closed-form solution

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

Saves us from doing the induction proof every time. We need this later

Mnemonics:

a: Branching (expansion, proliferation) factor for subproblems

b: Shrinkage factor for subproblem sizes

The missing c is for the hidden constant in $O(n^d \log^e n)$

Proof: Algorithms lecture, any good algorithms book, the script…

- Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms. 3rd edition. MIT Press, 2009
- Dasgupta, Papadimitriou, Vazirani: Algorithms. McGraw Hill, 2007
- Kleinberg, Tardos: Algorithm Design. Addison-Wesley, 2005
- Tim Roughgarden: Algorithms Illuminated. Soundlikeyourself Publishing, 2017

It is not as difficult as it may look, try yourself. See also AMP lecture

Note:
There are other versions of the Master Theorem, covering even more recurrences, and/or estimating constants. And theorems for other kinds of recurrences

Such things are clearly useful!

Example:

For the recursive Scan implementation, the Master Theorem applies to both work and depth:

- $W(1) = O(1)$
- $W(n) = W(n/2) + n$

a=1, b=2, d=1, e=0 gives $W(n) = O(n)$, Case 1 applies

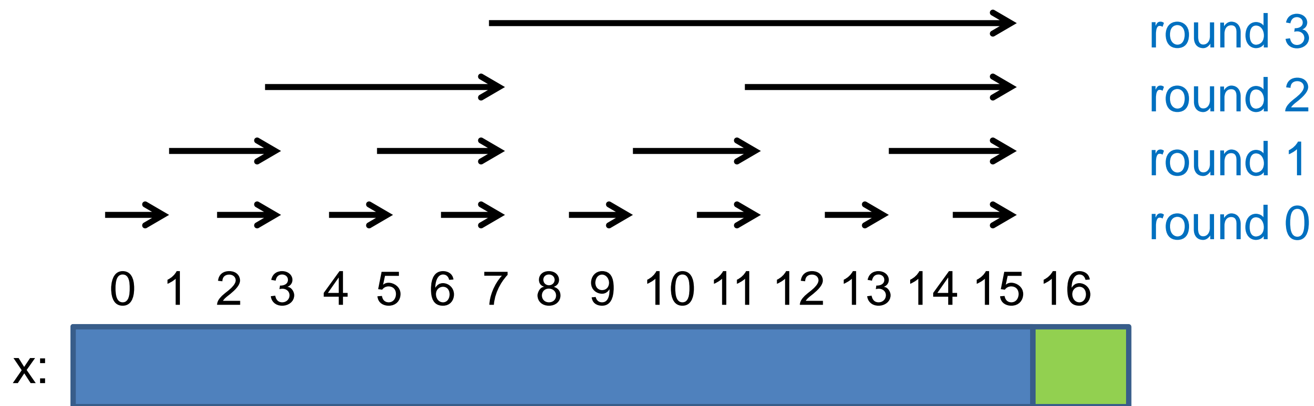
- $T(1) = O(1)$
- $T(n) = T(n/2) + 1$

a=1, b=2, d=0, e=0 gives $T(n) = O(\log n)$, Case 2 applies

More interesting use later

2. Iterative solution: Eliminate recursion and extra space

Perform $\log_2 n$ rounds, in round i, i≥0, a "+" operation is done for every 2^{i+1}'th element



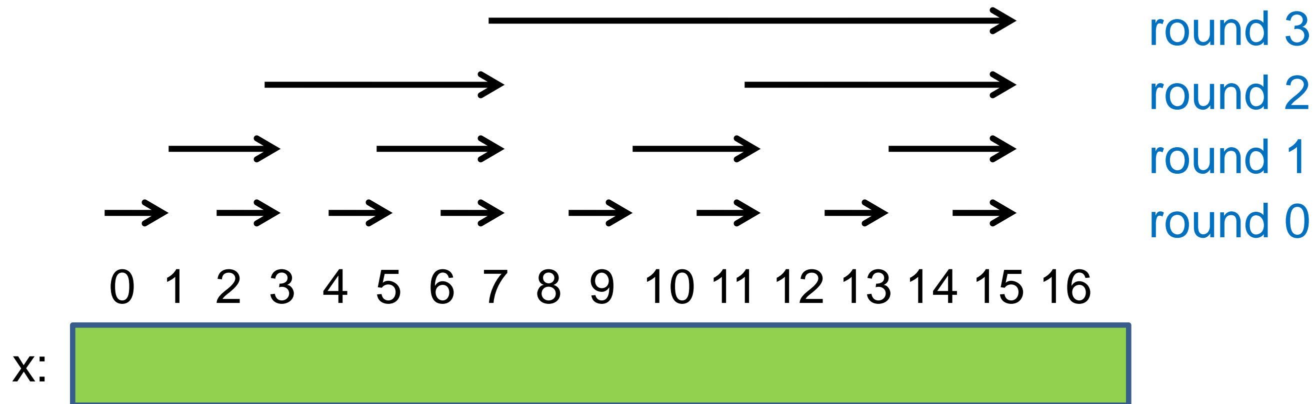
And almost done, now $x[2^k-1] = \sum_{0\leq i<2^k} x_i$

A synchronization operation (barrier) is needed after each round

Lemma:
Reduction can be performed out in $r = \log_2 n$ synchronized rounds, for n a power of 2. Total number of "+" operations are $n/2+n/4+n/8+\ldots<n$ $(=n-1)$

Recall, geometric series: $\sum_{0\leq i\leq n} ar^i = a(1-r^{n+1})/(1-r)$

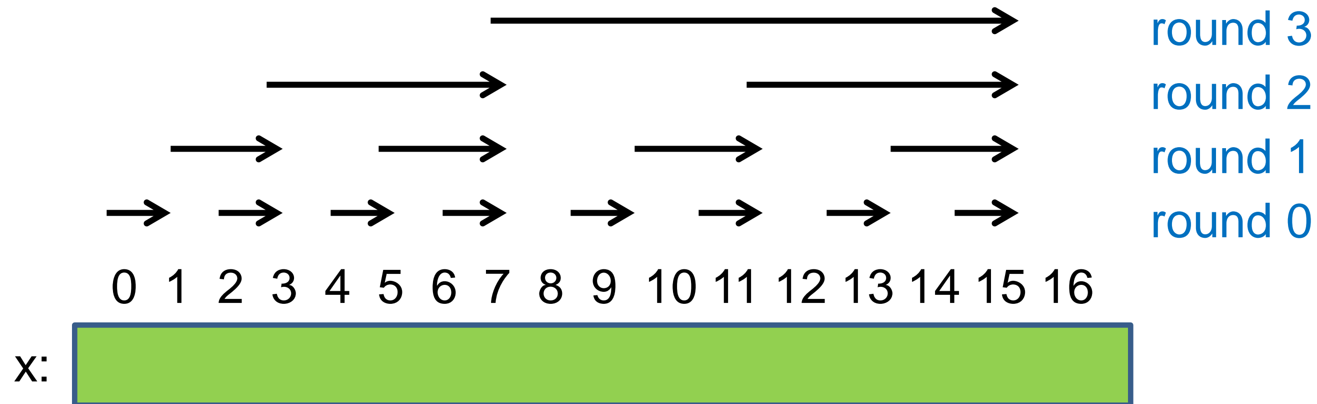
- Shared memory (programming) model: synchronization after each round
- Distributed memory programming model: ⟶ represent communication




```
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  for (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}
```

Data parallel computation, $n/2^{(k+1)}$ operations for round r, r=0, 1, …

Explicit synchronization

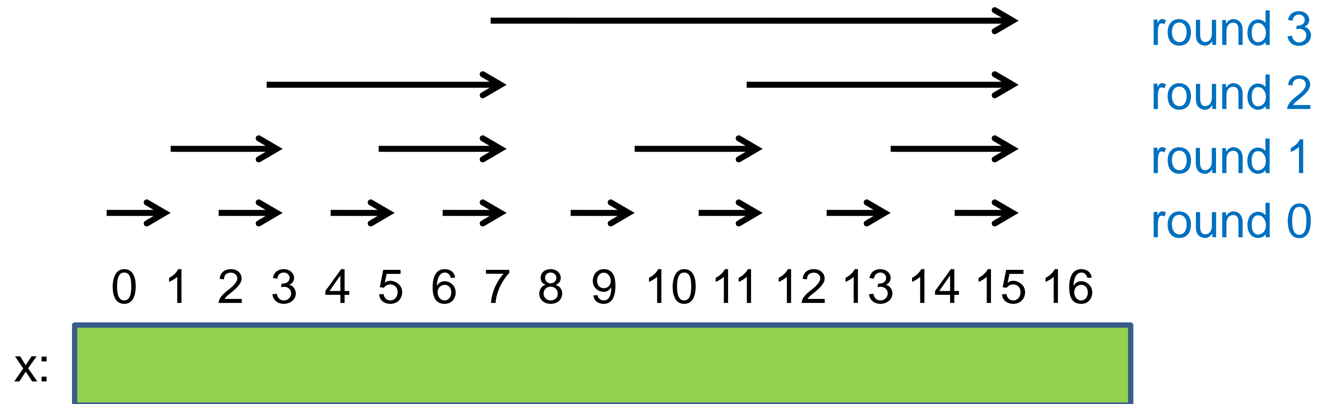



```
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  par (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}
```

Beware of dependencies

Data parallel computation, $n/2^{(k+1)}$ operations for round r, r=0, 1, …

Parallel Computing
TU WIEN Informatics




```
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  par (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}
```

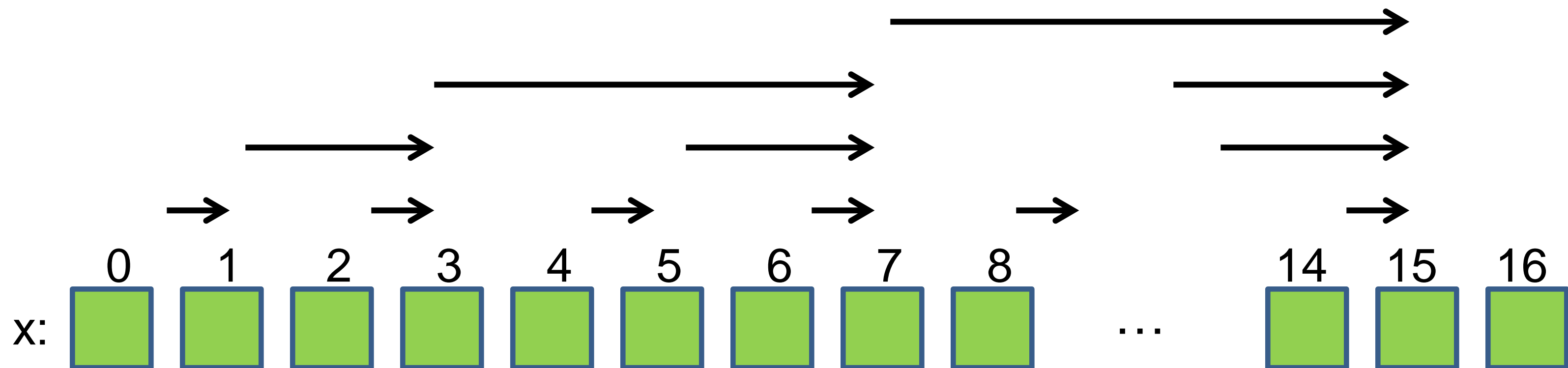
Data parallel computation, $n/2^{(k+1)}$ operations for round r, r=0, 1, …

But here are none. Why?

k is kk/2, so no update to x[i-k]

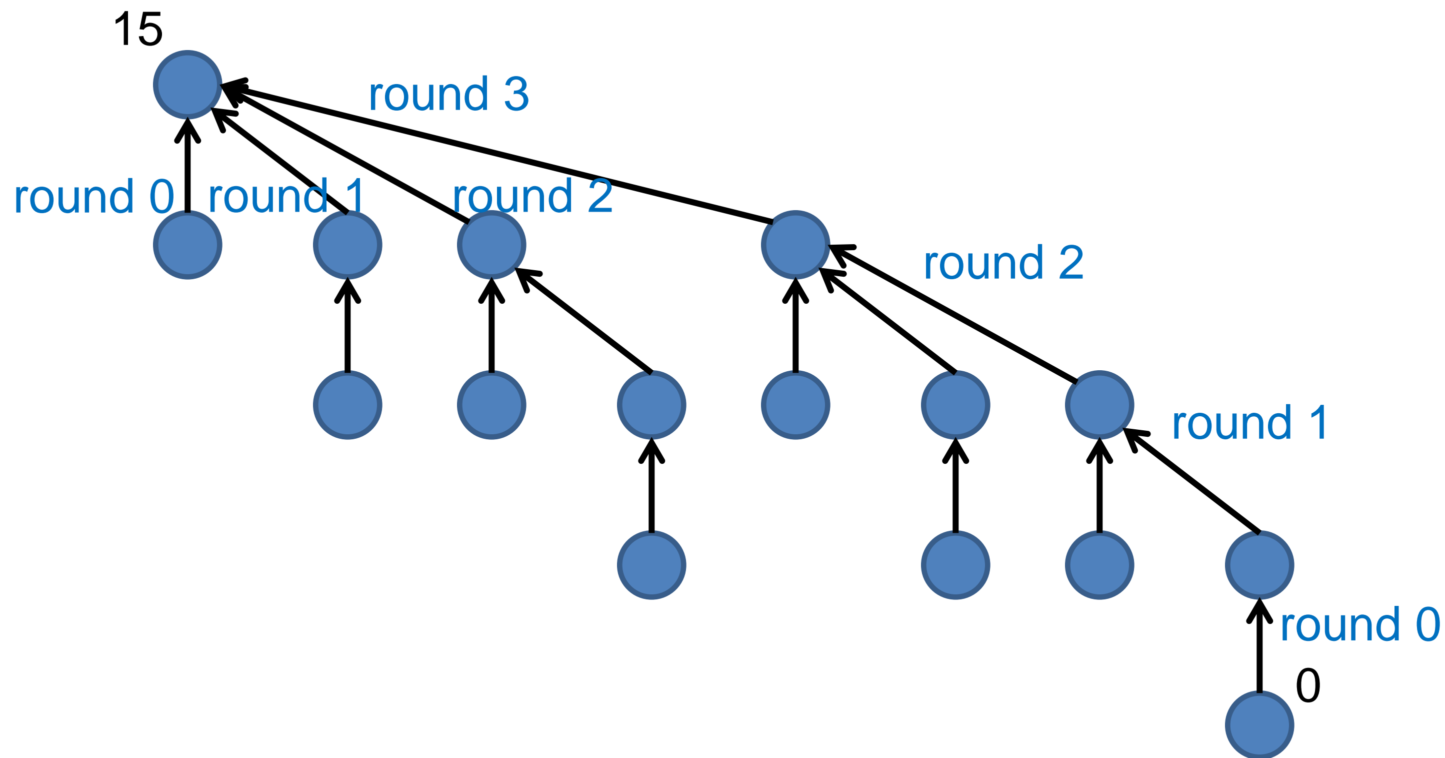
Distributed memory programming model

Prefix-sums problem solved with explicit communication: Message passing programming model

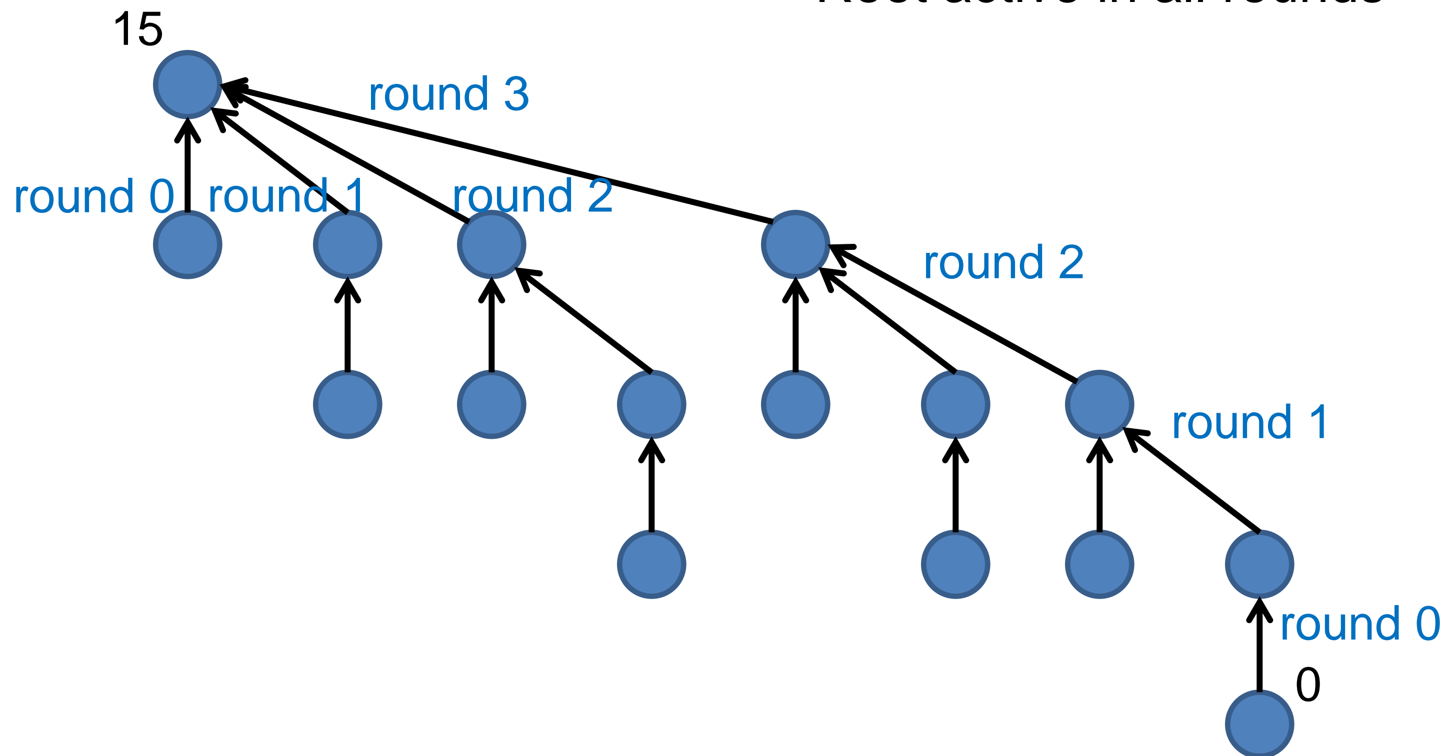


Beware: Much too costly for large n relative to p (see later what to do)

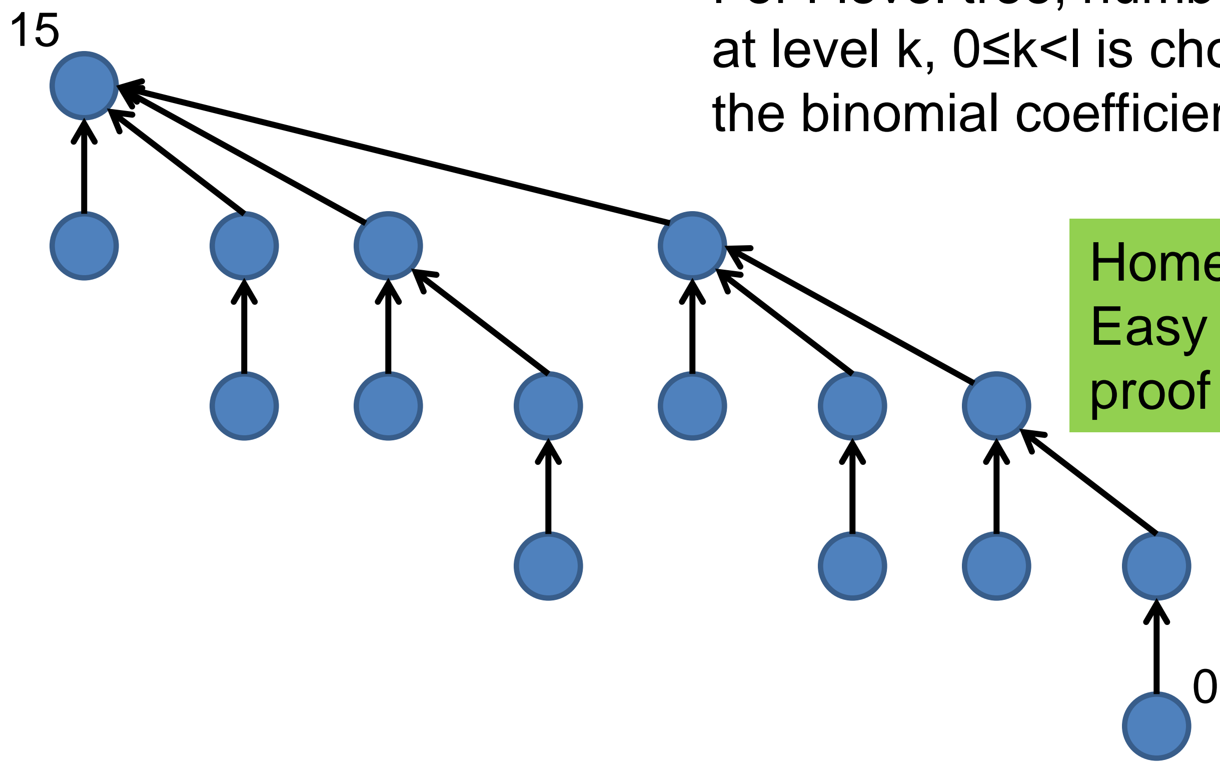
Communication pattern in distributed memory solutions



<u>Property 1</u>:
Root active in all rounds



This communication pattern is called Binomial Tree



Property 2:
For l-level tree, number of nodes at level k, 0≤k<l is choose(k,l-1), the binomial coefficient

Home exercise: Easy induction proof

This communication pattern is called Binomial Tree

So far, algorithm can compute the sum for arrays with $n=2^k$ elements for $k\geq 0$

- Repair for n not a power of 2?
- Extend to parallel prefix-sums?

Observation/invariant: let X be original content of array x before round k, $k=0,\ldots,\text{floor}(\log_2 n)$

$x[i] = X[i-2^k+1]+\ldots+X[i]$

for $i=j2^k-1$, $j=1, 2, 3, \ldots$

Idea:
Prefix sums computed for certain elements, use anther $\log_2 n$ rounds to extend partial prefix sums

Proof by invariant

```
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  par (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}
```

Invariant true before 0‘th iteration

If I true before k‘th iteration, must be true before (k+1)‘th

Invariant must imply conclusion/intended result

Home-work: prove correctness of this algorithm

©Jesper Larsson Träff

```
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  par (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}
```

Homework solution

In program, k doubles, so round number is log(k); do not to confuse with k in invariant

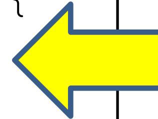
Invariant: X be original content of x. Before round k, k=0,…,floor($\log_2 n$), it holds that

$x[i] = \sum_{i-2^k+1 \leq j \leq i} X[j]$ for $i=j2^k-1$

Before round k=0, $x[i] = \sum_{i-2^k+1 \leq j \leq i} X[j] = \sum_{i \leq j \leq i} X[j] = X[i]$
True by definition, invariant holds before iterations start

Parallel Computing
TU WIEN Informatics

```
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  par (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}
```



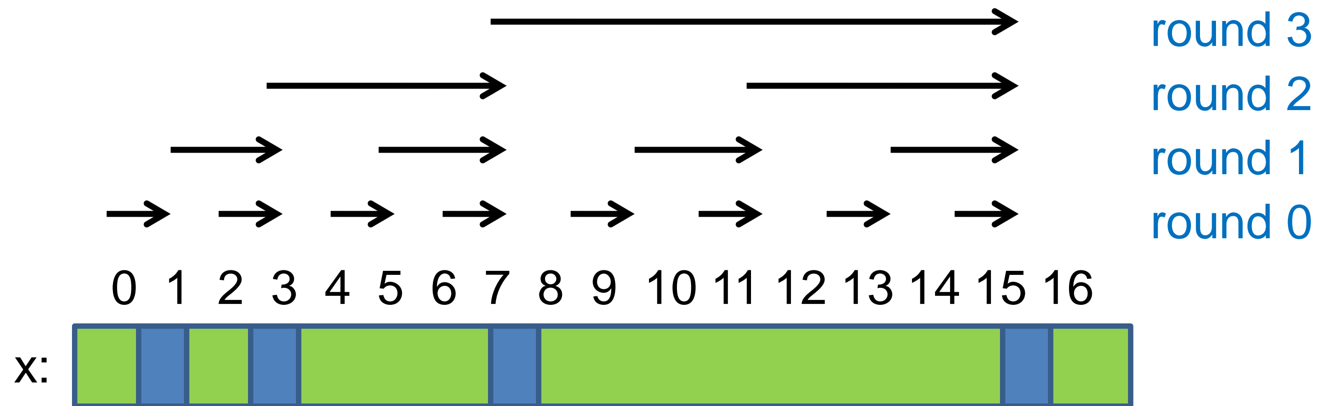
In program, k doubles, so round number is log(k); do not to confuse with k in invariant

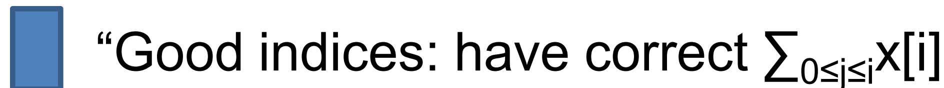
Invariant: X be original content of x. Before round k, k=0,…,floor($\log_2 n$), it holds that

$x[i] = \sum_{i-2^k+1\leq j\leq i} X[j]$ for $i=j2^k-1$

In round k, x[i] is updated by $x[i-2^k]+x[i]$. By the invariant this is $(\sum_{i-2^k-2^k+1\leq j\leq i-2^k} X[j]) + (\sum_{i-2^k+1\leq j\leq i} X[j]) = (\sum_{i-2^k-2^k+1\leq j\leq i} X[j]) = \sum_{i-2^{(k+1)}+1\leq j\leq i} X[j]$. The invariant therefore holds before the k+1'st iteration

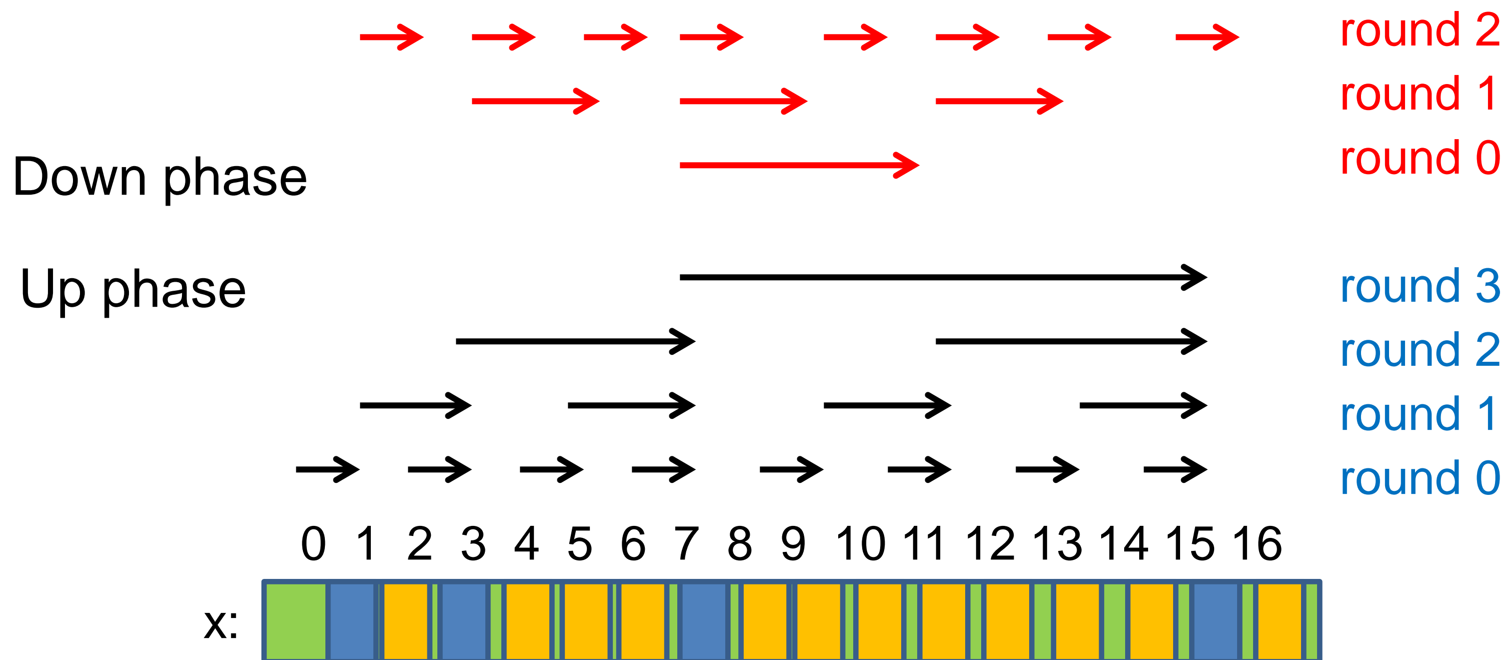
Extending to prefix-sums





Idea: Use another log n rounds to make remaining indices "good"

Extending to prefix-sums



Non-recursive, data parallel implementation

```
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  par (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}
```

"Up-phase":
$\log_2 n$ rounds,
$n/2+n/4+n/8+\ldots < n$
summations

These could be data dependencies, but are not

```
for (k=k>>1; k>1; k=kk) {
  kk = k>>1; // halve
  par (i=k-1; i<n-kk; i+=k) {
    x[i+kk] = x[i]+x[i+kk];
  }
  barrier;
}
```

"Down phase":
$\log_2 n$ rounds,
$n/2+n/4+n/8+\ldots < n$
summations

Total work ≈ 2n = O(Tseq(n))

But: factor 2 off from sequential n-1 work

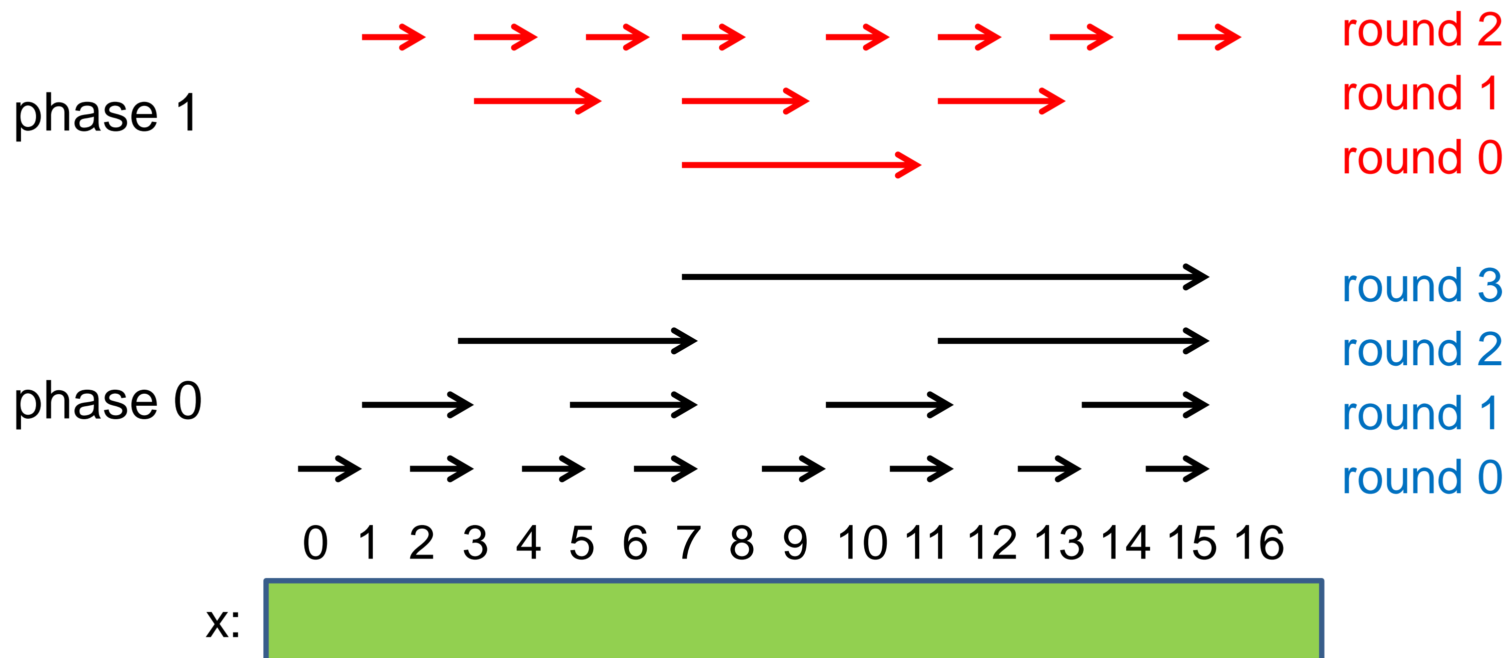
```
for (k=k>>1; k>1; k=kk) {
  kk = k>>1; // halve
  par (i=k-1; i<n-kk; i+=k) {
    x[i+kk] = x[i]+x[i+kk];
  }
  barrier;
}
```

Correctness: Need to prove that down-phase makes all indices good.

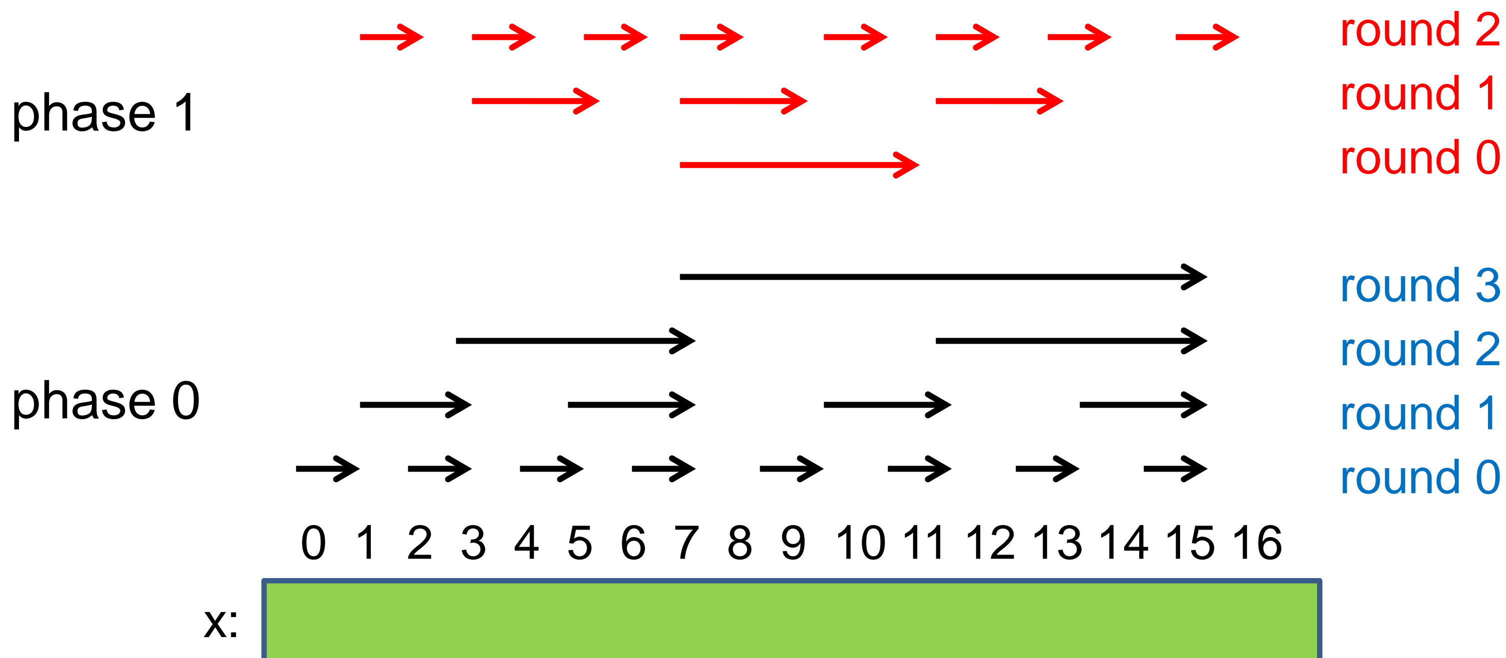
Prove invariant: $x[i] = \sum_{0\leq j\leq i} X[i]$, for $i=j2^k-1$, $j=1,2,3,\ldots$, $k=\text{floor}(\log n)$, $\text{floor}(\log_2 n)-1, \ldots$

Check at home

$S_p(n)$ at most p/2: Half the processors are “lost”



For p=n: Work optimal, but not cost optimal. The p processors are occupied in 2log p rounds = O(p log p)



2. Non-recursive solution: Summary

Advantages:

- In-place, needs no extra so space, input and output in same array
- Work-optimal, simple, parallelizable loops
- No recursive overhead

Drawbacks:

- Less cache-friendly than recursive solution, element access with increasing stride 2^k, less spatial locality (see later)
- 2 floor($\log_2 n$) rounds
- About 2n “+” operations

Prefix-sums on the PRAM

With some care, both recursive and non-recursive prefix-sums algorithms for the inclusive-prefix-sums problem can be implemented on the PRAM

Home exercise

<u>Theorem</u>:
The (inclusive/exclusive) prefix-sums problem for an array of n elements with an associative binary operator "+" can be solved on an EREW PRAM with p processors in O(n/p+log n) time steps.

With the blocking technique explained next, the result can be improved.

<u>Theorem</u>:
The (inclusive/exclusive) prefix-sums problem for an array of n elements with an associative binary operator "+" can be solved on an EREW PRAM with p processors in $O(n/p+\log p)$ time steps.

Aside: A lower bound/tradeoff for prefix-sums

Theorem (paraphrase): For computing the prefix sums for an n input sequence, the following tradeoff between "size" s (number of "+" operations) and "depth" t ($T\infty$) holds: $s+t \geq 2n-2$

Proof by examining "circuits" (model of parallel computation) that compute prefix sums

Roughly, this means: For fast parallel prefix sums algorithms, speedup (in terms of + operations) is at most p/2

Marc Snir: Depth-Size Trade-Offs for Parallel Prefix Computation. J. Algorithms 7(2): 185-201 (1986)
Haikun Zhu, Chung-Kuan Cheng, Ronald L. Graham: On the construction of zero-deficiency parallel prefix circuits with minimum depth. ACM Trans. Design Autom. Electr. Syst. 11(2): 387-409 (2006)

Prefix-sums for distributed memory models

Distributed memory programming model: ⟶ represents communication

Algorithm takes $2\log_2 n$ communication rounds, each with $n/2^k$ concurrent communication operations, total of 2n communication operations. Since often n»p, much too expensive

Blocking technique:

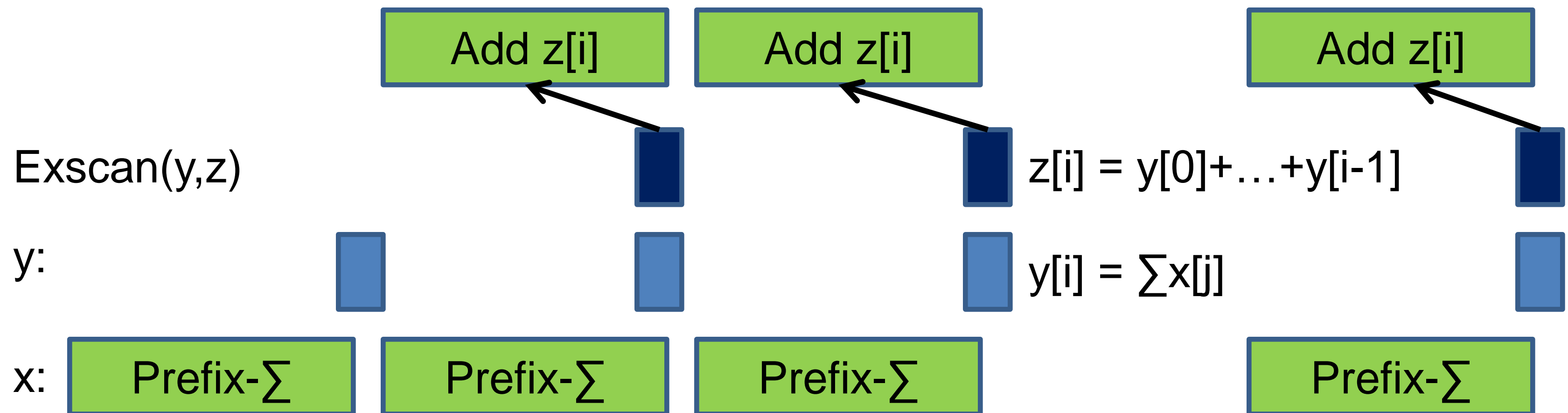
Reduce number of communication/synchronization steps by dividing problem into p similar, smaller problems (of size n/p) that can be solved sequentially (in parallel), apply parallel algorithm on selected element from p blocks

Blocking technique for prefix-sums algorithms

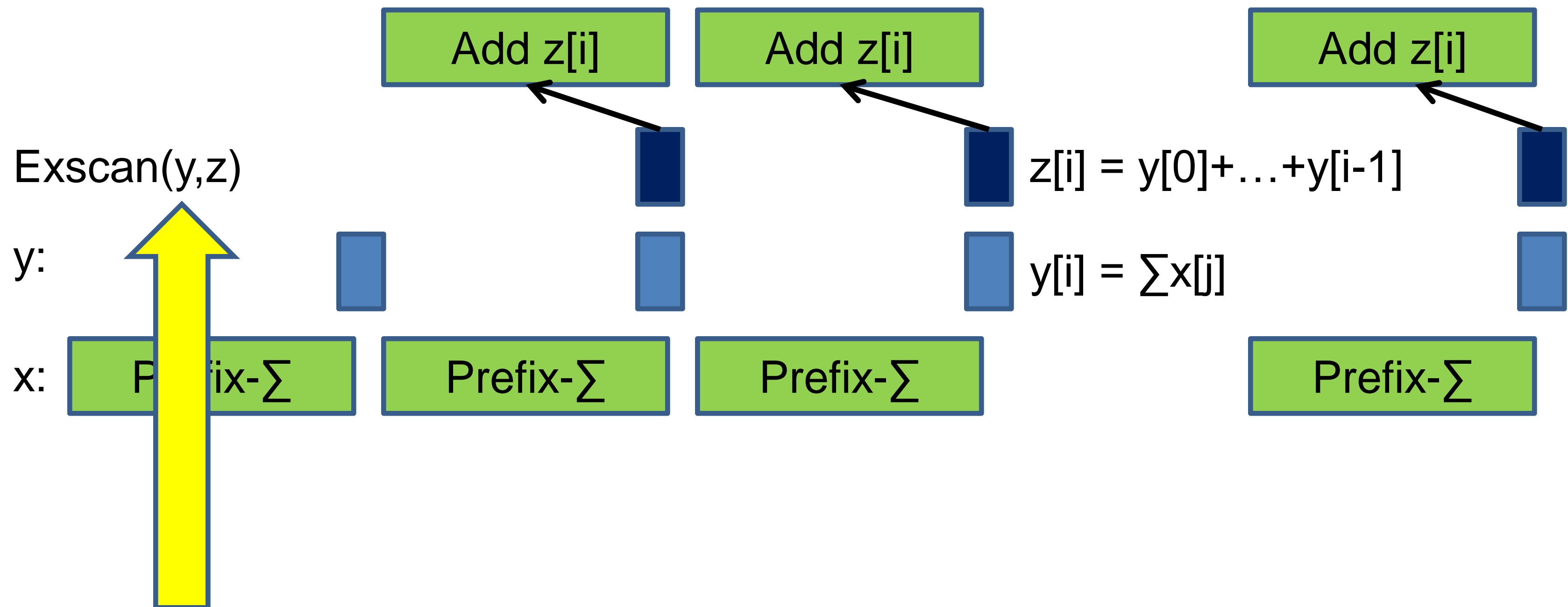
1. Processor i has block of n/p elements, x[i n/p,…,(i+1)n/p-1]
2. Processor i computes prefix sums of x[j], total sum in y[i]
3. Exscan(y,p);
4. Processor i adds exclusive prefix sum y[i] to all x[i n/p,…,(i+1)n/p-1]

Processors locally, without synchronization, compute prefix-sums on local (part of) array of size n/p (block). Exscan (exclusive prefix-sums) operation takes O(log p) communication rounds/synchronization steps, and O(p) work. Processors complete by local postprocessing (block).

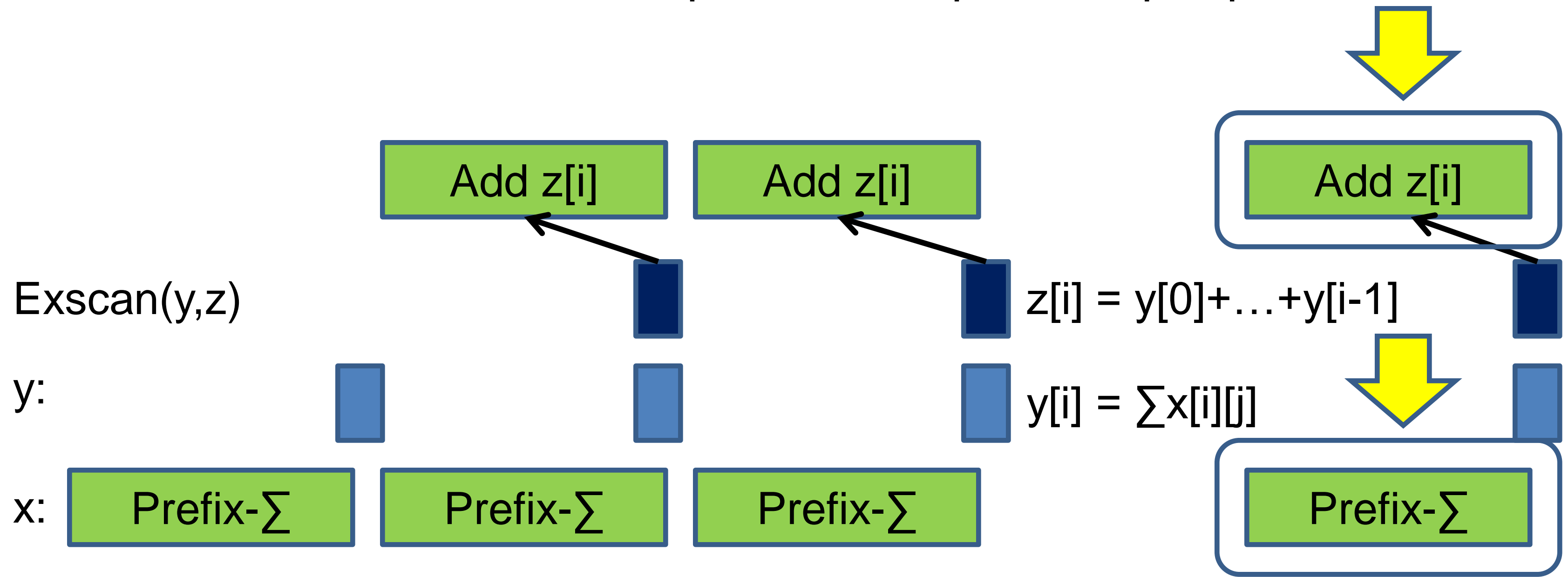
Observation:
Total work ("+" operations) is 2n+p log p; at least twice Tseq(n)


Add z[i]
Add z[i]
Add z[i]
Exscan(y,z)
z[i] = y[0]+…+y[i-1]
y:
y[i] = ∑x[j]
x:
Prefix-∑
Prefix-∑
Prefix-∑
Prefix-∑

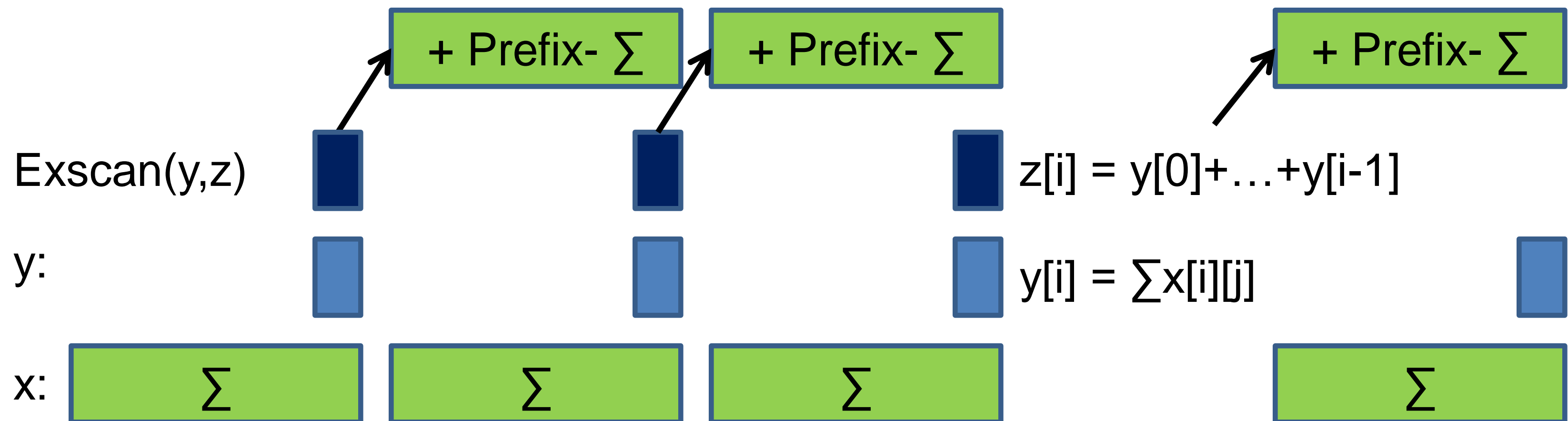
Note: This is not the best possible application of the blocking technique (hint: Better to divide into p+1 parts)



After solving local problems on blocks: p elements, p processors. Algorithm that is as fast as possible (small number of rounds) needed, does not need to be work optimal!


Sequential computation per processor
Add z[i]
Add z[i]
Add z[i]
Exscan(y,z)
z[i] = y[0]+…+y[i-1]
y:
y[i] = ∑x[i][j]
x:
Prefix-∑
Prefix-∑
Prefix-∑
Prefix-∑

Observation: Possibly better by reduction first, then prefix-sums



Naïve (per block) analysis:

- Prefix first: 2n read, 2n write operations (per block)
- Reduction first: 2n read operations, n write operations
- Both: ≥2n-1 "+" operations

Blocking technique summary

Technique:

1. Divide problem into p roughly equal sized parts (subproblems)
2. Assign subproblem to each processor
3. Solve subproblems using sequential algorithm
4. Use parallel algorithm to combine subproblem results
5. Apply combined result to subproblem solutions

Analysis:

1-2: Should be fast, O(1), O(log n), …

3: perfectly parallelizable, e.g. O(n/p)

4: Should be fast, e.g., O(log p), total cost must be less than O(n/p)

5: Perfectly parallelizable

Blocking technique summary

Technique:
1. Divide problem into p roughly equal sized parts (subproblems)
2. Assign subproblem to each processor
3. Solve subproblems using sequential algorithm
4. Use parallel algorithm to combine subproblem results
5. Apply combined result to subproblem solutions

Use when applicable

BUT blocking is not always applicable!

Examples:
- Prefix-sums – data independent
- Cost-optimal merge – data dependent

Step 1 quite non-trivial

Blocking technique: Another view

Technique:

1. Use work-optimal algorithm to shrink problem enough
2. Use fast, possibly non work-optimal algorithm on shrunk problem
3. Unshrink, compute final solution with work-optimal algorithm

Remark:

1. Typically from O(n) to O(n/log n) using O(n/log n) processors
2. Use O(n/log n) processors on O(n/log n) sized problem in O(log n) time steps

1. Processor i has block of n/p elements, x[i n/p,…,(i+1)n/p-1]
2. Processor i computes prefix sums of x[j], total sum in y[i]
3. Exscan(y,p);
4. Processor i adds exclusive prefix sum y[i] to all x[i n/p,…,(i+1)n/p-1]

Complexity:

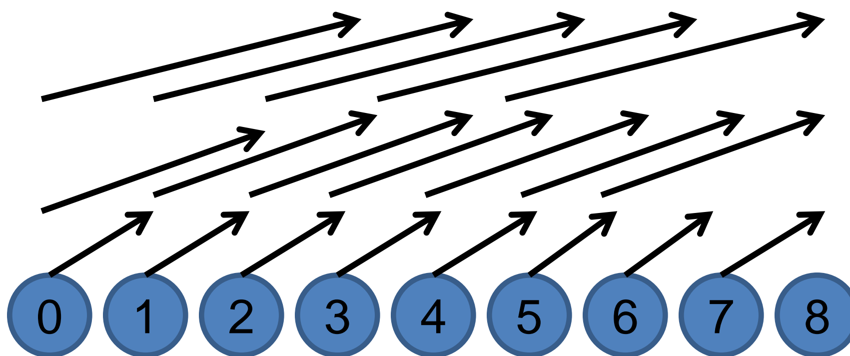
1. O(1)
2. T = n/p, W = n
3. T = O(n/p), W = O(n) with p procesors (e.g. O(log p) for p in O(n/log n))
4. T = n/p, W = n

If conditions in Step 3 fulfilled, blocked prefix-sums algorithm is work-optimal. Use fastest possible prefix-sums with work not exceeding O(n)

3. Yet another data parallel prefix-sums algorithm: Doubling

Idea: In each round, let each processor compute sum x[i] = x[i-k]+x[i] (not only every k'th processor, as in Solution 2)

Round k', $k=2^{k'}$



Only ceil($\log_2 n$) rounds needed, almost all processors active in each round; correctness by similar argument as solution 2

W. Daniel Hillis, Guy L. Steele Jr.: Data Parallel Algorithms. Commun. ACM 29(12): 1170-1183 (1986)

3. Yet another data parallel prefix-sums algorithm: Doubling

```
for (k=1; k<n; k<<=1) {
  for (i=k; i<n; i++) x[i] = x[i-k]+x[i];
  barrier;
}
```

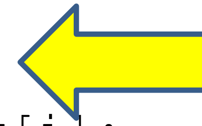
Data parallel?

Why might it not be correct?
There are dependencies

- How could this implementation be correct? Why? Invariant?

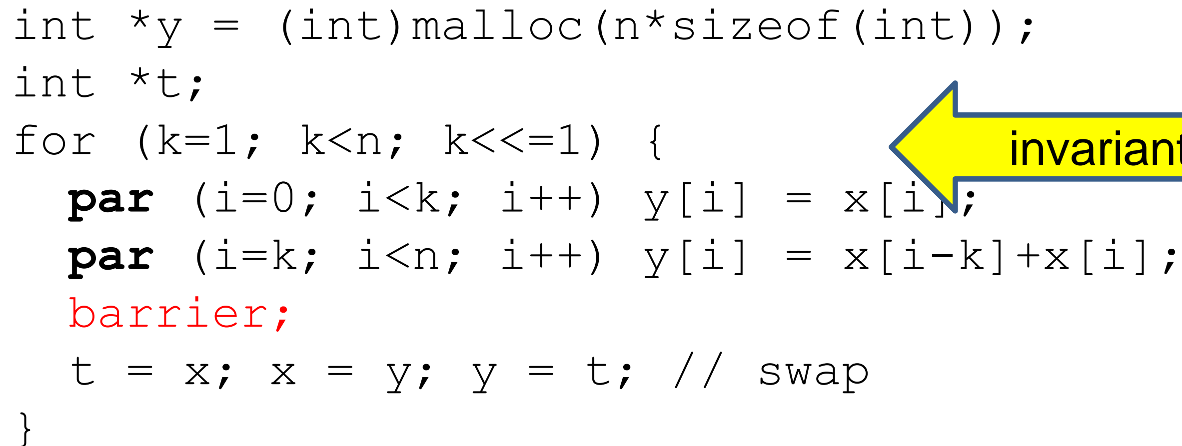
- All indices i≥1 active in each rounds, total work O(n log n)
- But only log n rounds

```
int *y = (int)malloc(n*sizeof(int));
int *t;
for (k=1; k<n; k<<=1) {
  par (i=0; i<k; i++) y[i] = x[i];
  par (i=k; i<n; i++) y[i] = x[i-k]+x[i];
  barrier;
  t = x; x = y; y = t; // swap
}
```



Eliminate dependencies with extra array. Both loops now data parallel

Iterations in update-loop not independent, thus loop not immediately parallelizable: Eliminate dependencies




```
int *y = (int)malloc(n*sizeof(int));
int *t;
for (k=1; k<n; k<<=1) {
  par (i=0; i<k; i++) y[i] = x[i];
  par (i=k; i<n; i++) y[i] = x[i-k]+x[i];
  barrier;
  t = x; x = y; y = t; // swap
}
```

Invariant:
before iteration step k, $x[i] = \sum_{\max(0,i-2^k+1)\leq j\leq i} X[j]$ for all i

It follows that the algorithm solves the inclusive prefix-sums problem

3. Doubling prefix-sums algorithm: Summary

Advantages:

- Only ceil($\log_2 p$) rounds (synchronization/communication steps)
- Simple, parallelizable loops
- No recursive overhead

Drawbacks:

- NOT work-optimal
- Less cache-friendly than recursive solution, element access with increasing stride k, less spatial locality (see later)
- Extra array of size n needed to eliminate dependencies

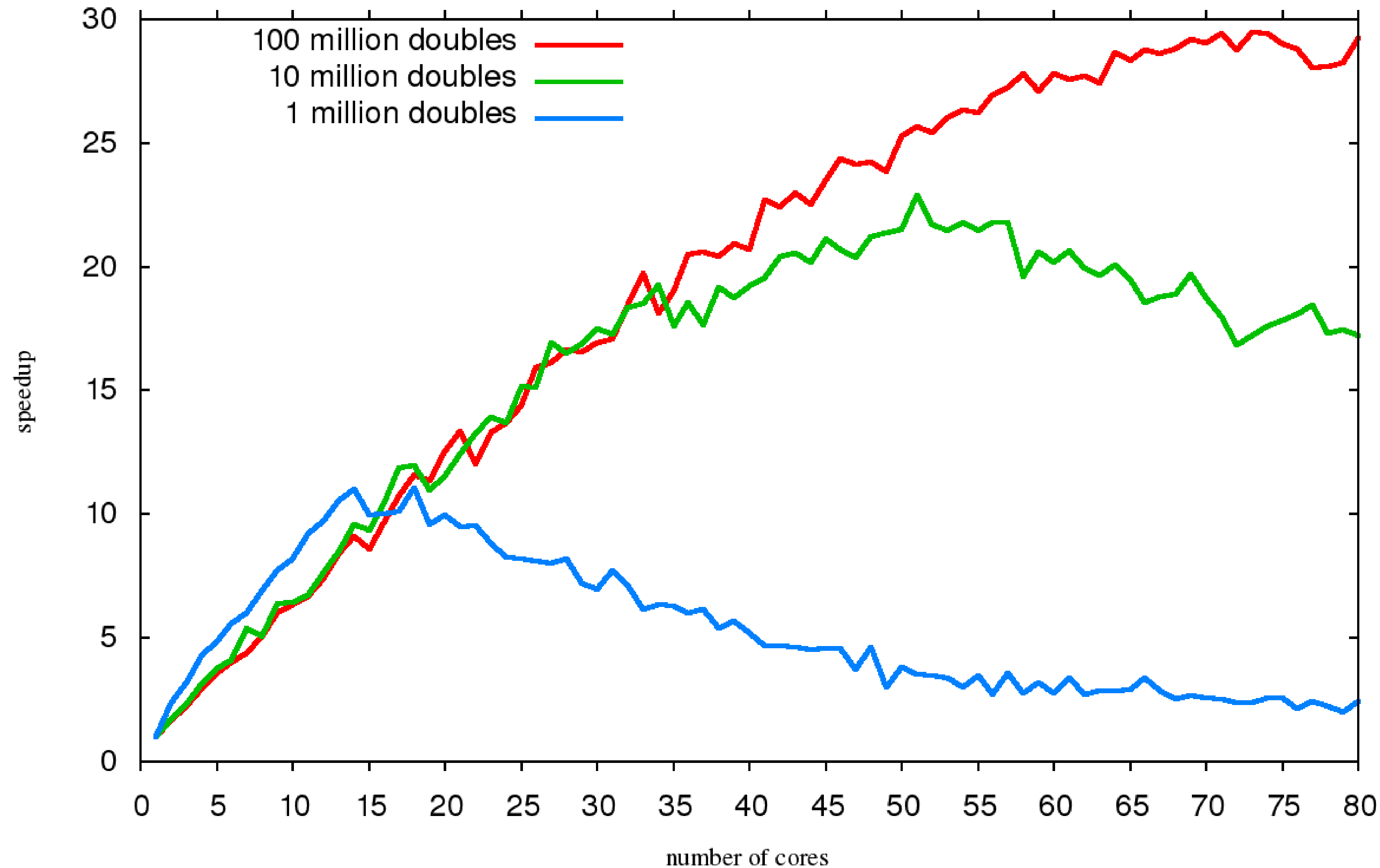
©Jesper Larsson Träff

Some results:

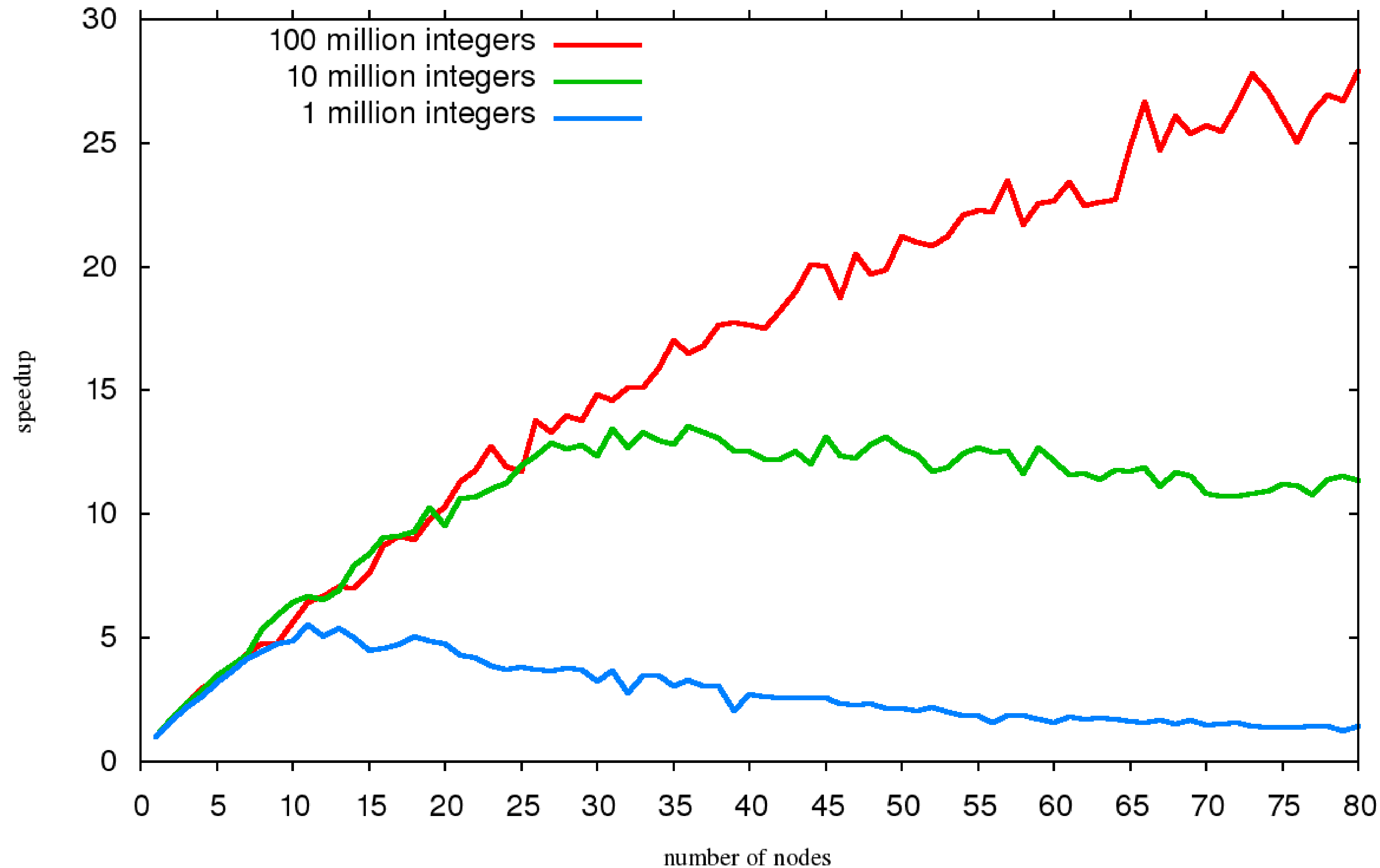
We implemented (ca. 2015) prefix-sums algorithms (with some extra optimizations, and not exactly as just described), and executed on

- saturn: 48-core AMD system, 4*2*6 NUMA-cores, 3-level cache
- mars: 80-core Intel system, 8*10 NUMA-cores, 3-level cache

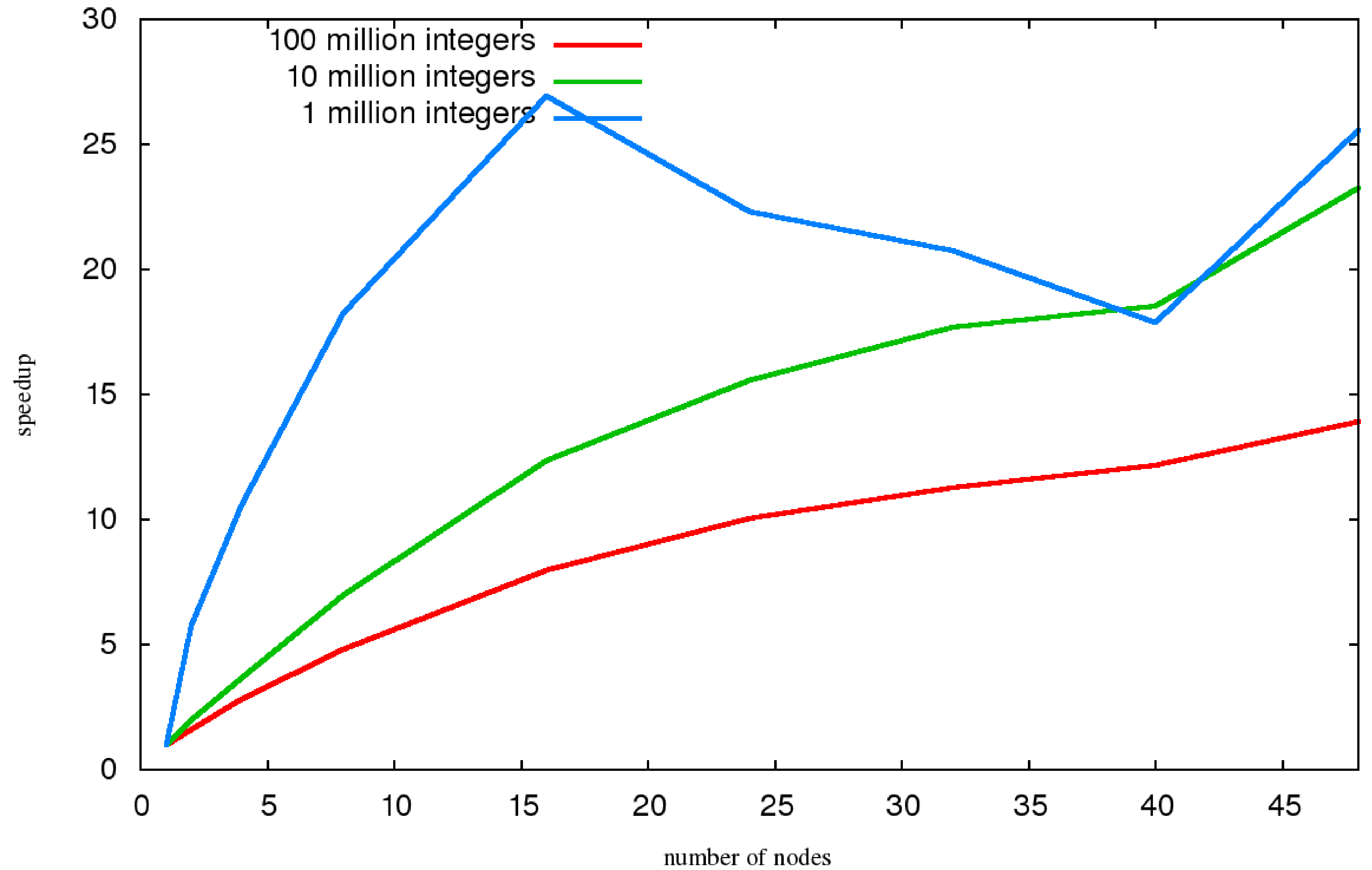
and computed speedup relative to a good (but probably not “best known/possible”) sequential implementation (25 repetitions of measurement)



mars, basetype double

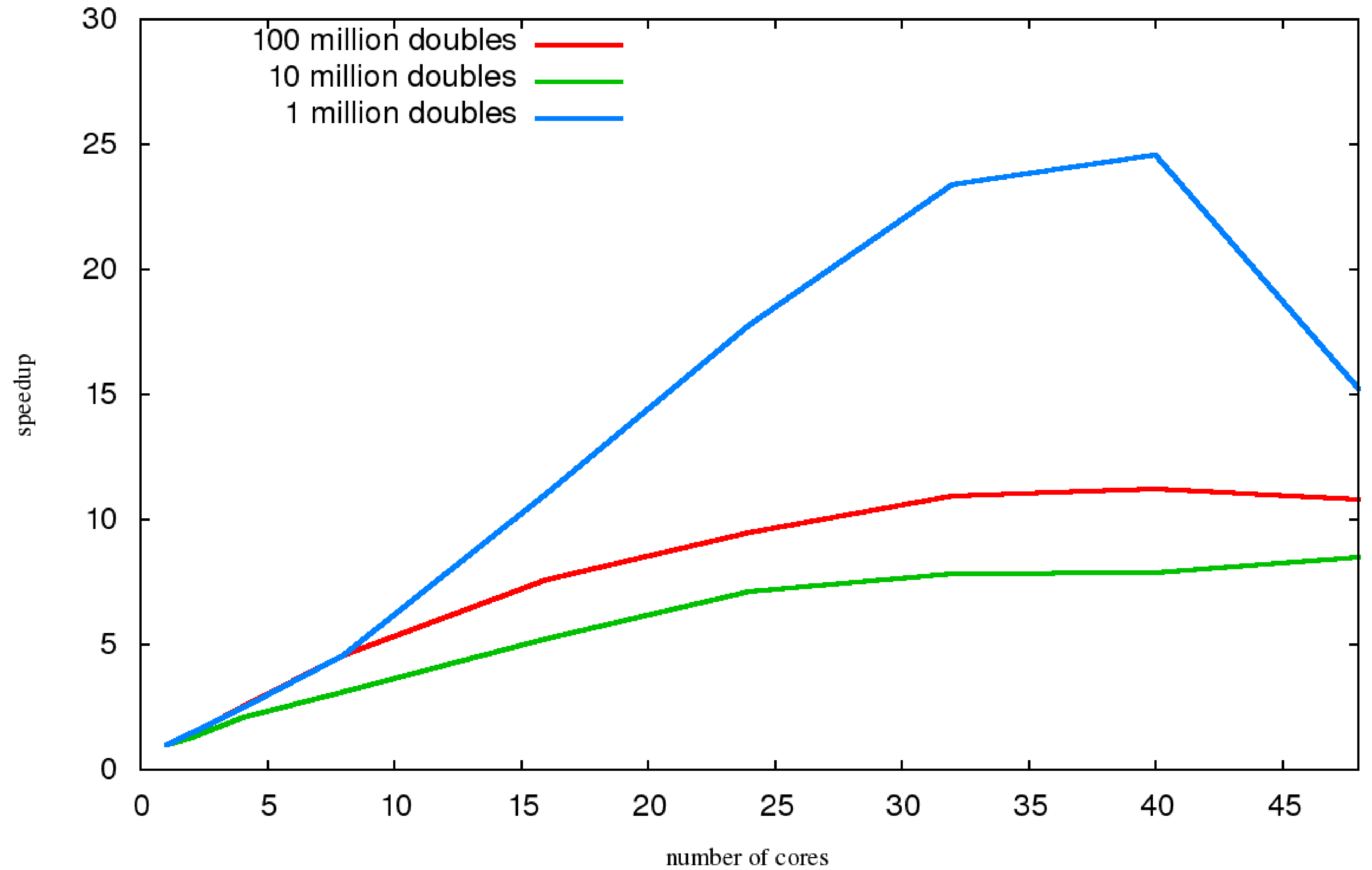


mars, basetype int



saturn, basetype int

©Jesper Larsson Träff



saturn, basetype double

A generalization of the prefix-sums problem: List-ranking

Given list x_0 -> x_1 -> x_2 -> … -> x_{n-1}, compute all n list-prefix sums

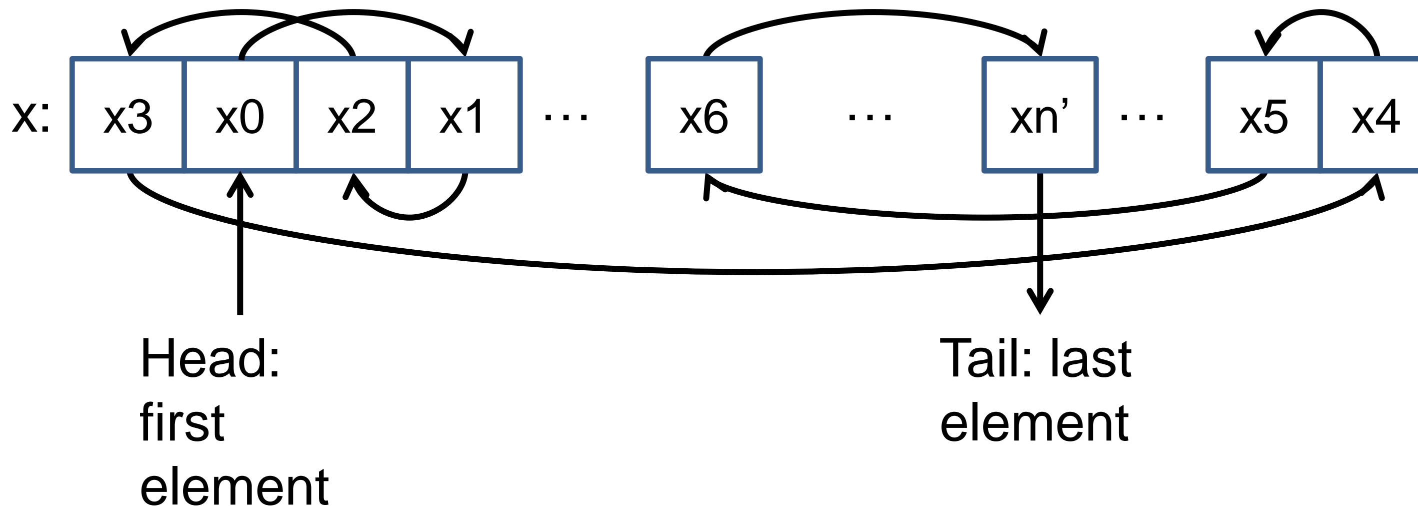
$$y_i = x_i+x_{i+1}+x_{i+2}+\ldots+x_{n-1}$$

by following -> from x_i to end of list, "+" some associative operation on type of list elements

Sequentially, looks like an easy problem (similar to prefix sums problem): Find list head, follow the pointers and sum up, O(n)

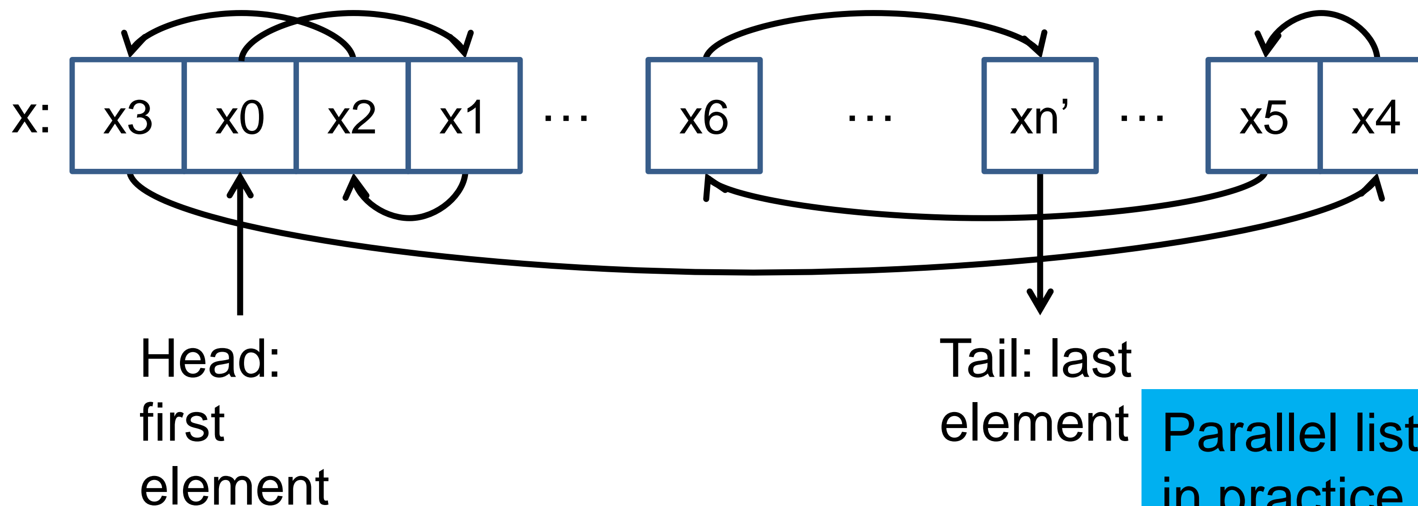
The list ranking problem is to compute, for each list element, the sum of the values of all following elements. The problem is sometimes called the "data-dependent prefix sums problem"

Standard assumption: List stored in array



- Input compactly in array, index of first element may or may not be known
- Index of element in array has no relation to position in list

Standard assumption: List stored in array

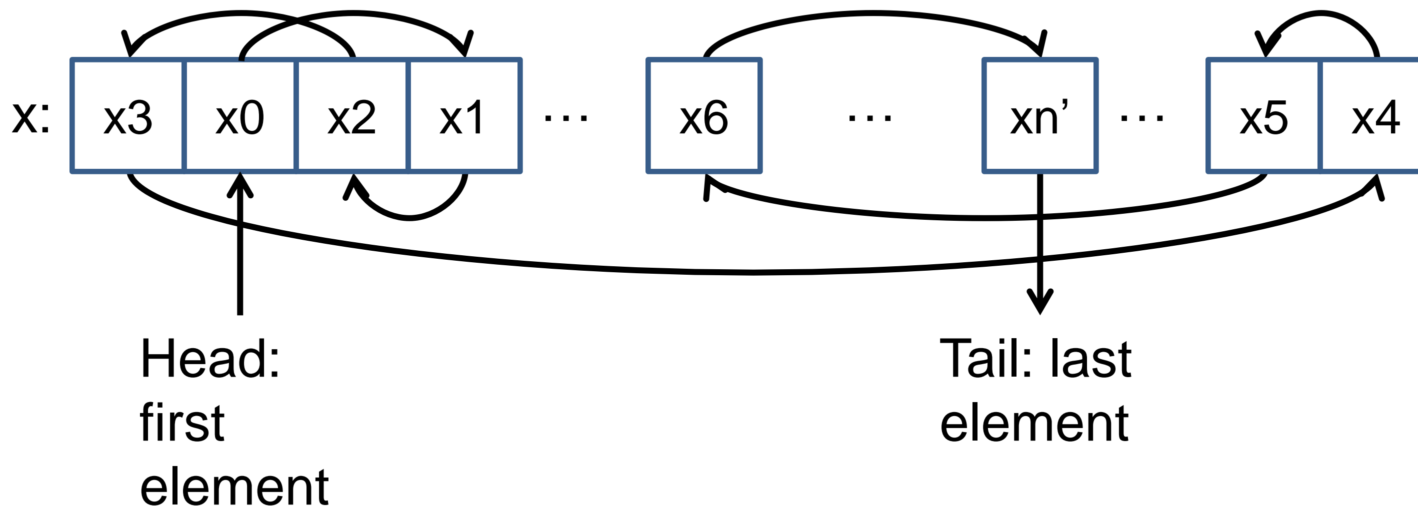


Parallel list ranking in practice can work for very large n

A difficult problem for parallel computing: What are the subproblems that can be solved independently?

Major, theoretical result (PRAM model): The list ranking problem can be solved in O(n/p+log n) parallel time steps.

Standard assumption: List stored in array

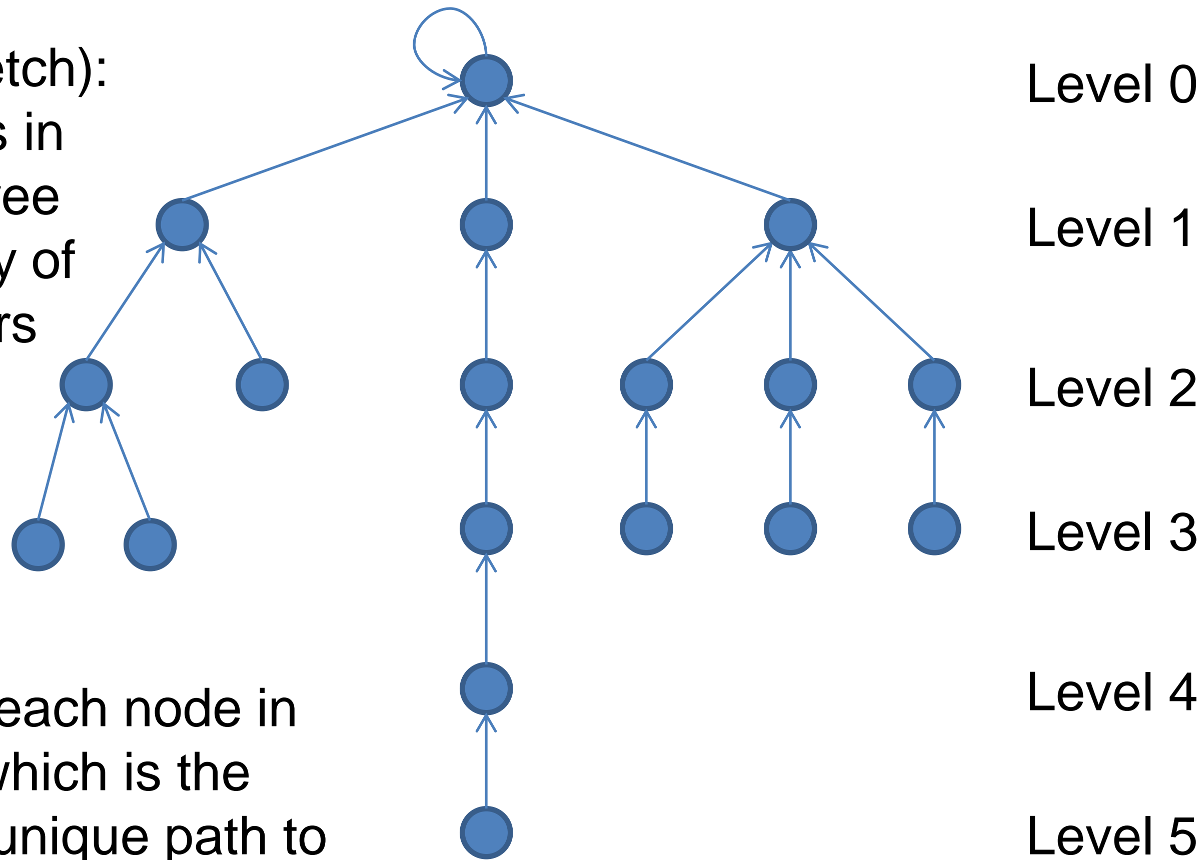


Richard J. Anderson, Gary L. Miller: Deterministic Parallel List Ranking. Algorithmica 6(6): 859-868 (1991)

Major, theoretical result (PRAM model): The list ranking problem can be solved in O(n/p+log n) parallel time steps.

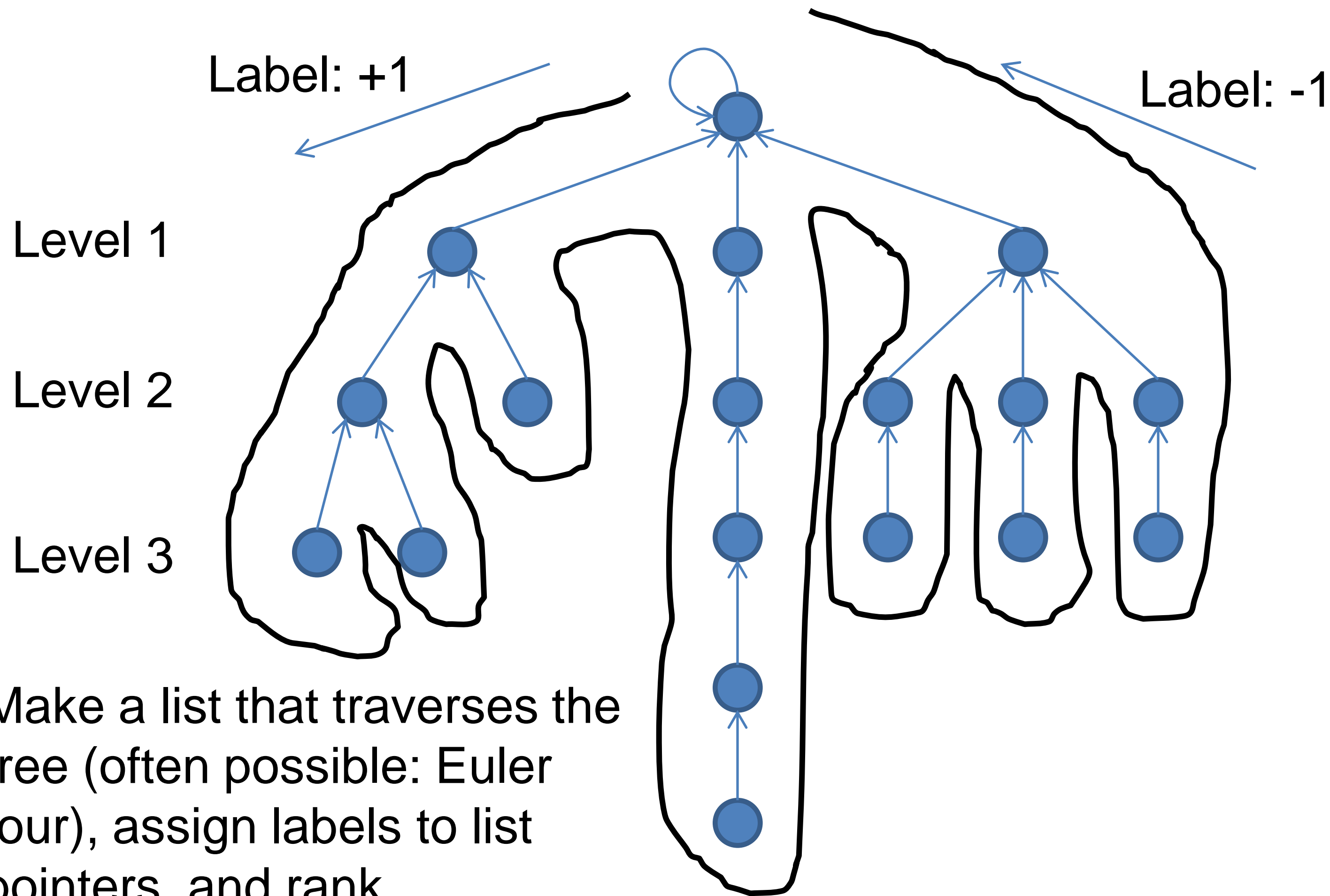
Usefulness of list ranking: Tree computations

Example (sketch): Finding levels in rooted tree, tree given as array of parent pointers



Task: Assign each node in tree a level, which is the length of the unique path to the root

Usefulness of list ranking: Tree computations



Make a list that traverses the tree (often possible: Euler tour), assign labels to list pointers, and rank

Technique for problem partitioning: Blocking

Linear time problem with input in n-element array, p processors. Divide array into p independent, consecutive blocks of size $\Theta(n/p)$ using $O(f(p,n))$ time steps per processor. Solve p subproblems in parallel, combine into final solution using $O(g(p,n))$ time steps per processor

The resulting parallel algorithm is cost-optimal if both $f(p,n)$ and $g(p,n)$ are $O(n/p)$

Examples:

- Prefix-sums, partition in $O(1)$ time
- Merging, partition in $O(\log n)$ time

List-ranking problem not really solvable by blocking

Other problems for parallel algorithmics

Versatile operations with simple, practical, sequential algorithms and implementations; that are (extremely)difficult to parallelize well, in theory and practice:

Graph search, G=(V,E) (un)directed graph with vertex set V, n=|V| and edge set E, m=|E|, some given source vertex v in V:

- Breadth-first search (BFS) ← Hard, graph structure dependent
- Depth-first search (DFS) ← Really Hard; perhaps impossible
- Single-source shortest path (SSSP)
- Transitive closure
- …

Lesson: Building blocks from sequential algorithmics are highly useful for parallel computing algorithms; but not always

Reduction, Scan: Other collective operations

A set of processors collectively carry out an operation in cooperation on sets of data:

- Broadcast: One processor has data, after operation all processors have data
- Scatter: Data of one processor distributed in blocks to other processors
- Gather: Blocks from all processors collected at one processor
- Allgather: Blocks from all processors collected at all processors (aka Broadcast-to-all)
- Alltoall: Each processor has blocks of data, one block for each other processor, each processor collects blocks from other processor

Lecture summary, checklist

- Merging in parallel (4 algorithms)
- Bitonic merge
- Prefix sums problem (4 algorithms)
- Blocking technique
- Prefix sums for load balancing and processor allocation
- List ranking, BFS, DFS, SSSP: Difficult to parallelize problems