Parallel Algorithms and Parallel Computers (ii)

IN4026

Cees Witteveen

C.Witteveen@tudelft.nl

Algorithmics Group

Sorting methods

- sorting by merging
  - lowerbound for sorting
  - naive parallel sorting by merging
  - improved parallel sorting by merging
  - parallel sorting by fast merging
- enumeration sort
- random quicksort

Lower bound for sorting

- Given a sequence X of *n* input values.
  There are $n! = n \times (n-1) \times (n-2) \times \ldots \times 1$ possible outcomes when sorting X.

- A sequential sorting algorithm A can be conceived as a procedure to generate a *permutation* of its input. Every permutation of its input is possible. If we model such an algorithm as a *decision tree.* Such a decision tree is a tree with *n!* leaves given an input of size *n.*

- If we assume that the algorithm can only make *binary comparisons* between values $x < y$ or $x \geq y$, t tree.

comparison-based:
heapsort, quicksort, bubble sort, insertion sort

- A binary tree with *n!* leaves has a minimum depth of $\log_2 n!$
  We now that $\log_2 n! = \Omega(n \log n)$.

- Therefore the time complexity of any comparison-based sequential sorting algorithm A is $\Omega(n \log n)$.

Lower bound for sorting

Deriving the lower bound:

Note that

$n! \geq n(n-1)\ldots(n/2) \geq (n/2)^{n/2}$.

Hence,

$$\begin{aligned}\log n! &\geq \log (n/2)^{n/2} \geq n/2 \log n/2 \\ &\geq n/2 (\log n - \log 2) \\ &\geq n/2 \log n - n/2 \log 2 \\ &\geq n/2 \log n - n/4 \log n \\ &\geq n/4 \log n\end{aligned}$$

for $n \geq 4^{\log 2} = 4$

Therefore,

$\log n! = \Omega(n \log n)$

sorting by merging

There is an efficient sequential sorting by merging algorithm taking *O(n log n)* time to sort an array A[1..n]:

```
mergesort(A,n)
begin
    if n =1 then return A[1]
    else
        A1 = mergesort(A[1..n/2],n/2);
        A2 = mergesort(A[n/2+1..n],n/2);
        merge(A1, A2, n/2, n/2);              % merges two sorted sequences
end
```

merge is a $T(m,n) = O(m+n)$ algorithm for merging two sorted sequences

(Essential idea: use two pointers p_1, p_2 for array A and B, respectively. Starting with $p_1 = p_2 = 1$, if $A[p_1] \leq B[p_2]$, write $A[p_1]$ as the next element of C and set $p_1 := p_1+1$, else write $A[p_2]$ as the next element of C and set $p_2 := p_2+1$)

sorting by merging

There is an efficient sequential sorting by merging algorithm taking *O(n log n)* time to sort an array A[1..n]:

```
mergesort(A,n)
begin
    if n =1 then return A[1]
    else
        A1 = mergesort(A[1..n/2],n/2);
        A2 = mergesort(A[n/2+1..n],n/2);
        merge(A1, A2, n/2, n/2);
end
```

$T(1) = O(1)$

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

$\boxed{T(n) = O(n \log n)}$

merge is a $T(m,n) = O(m+n)$ algorithm for merging two sorted sequences

(Essential idea: use two pointers p_1, p_2 for array A and B, respectively. Starting with $p_1 = p_2 = 1$, if $A[p_1] \leq B[p_2]$, write $A[p_1]$ as the next element of C and set $p_1 := p_1+1$, else write $A[p_2]$ as the next element of C and set $p_2 := p_2+1$)

naive parallel sorting by merging

There is a *naive* parallel sorting method by parallelizing the recursive step of the algorithm:

naive_par_mergesort(A,n)
begin

1. if n =1 then return A[1] — $T = O(1)$; $W = O(1)$

2. for $1 \le i \le 2$ pardo — $T = T(n/2)$; $W = 2W(n/2)$

   A_i = naive_par_mergesort($[a_{1+n/2.(i-1)}, a_{2+n/2.(i-1)}, \ldots a_{n/2+n/2.(i-1)}]$);

3. merge(A_1, A_2, n/2, n/2); — $T = O(n)$; $W = O(n)$

end

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

naive parallel sorting by merging

There is a *naive* parallel sorting method by parallelizing the recursive step of the algorithm:

naive_par_mergesort(A,n)
begin

1. if n =1 then return A[1]

2. for $1 \leq i \leq 2$ pardo

   A_i = naive_par_mergesort($[a_{1+n/2.(i-1)}, a_{2+n/2.(i-1)}, \ldots a_{n/2+n/2.(i-1)}]$);

3. merge(A_1, A_2, n/2, n/2);

end

T = O(1) ; W = O(1)

T = T(n/2) ; W = 2W(n/2)

T = O(n) ; W = O(n)

T = O(n) ; W = O(n log n)

We can do better by trying to parallelize the costly merging step as well !

Merging of sorted sequences

- Problem

  Let A[1 .. m] and B[1 .. n] be sorted sequences.

  We want to compute the *sorted merge* C[1..m+n] of A and B by using a parallel algorithm.

- Observation

  There is a (best) sequential algorithm for merging with $T^*(m,n) = O(m+n)$.

- Consequence

  A (weakly) optimal parallel algorithm has to achieve $W(m,n) = O(m+n)$.

Merging of sorted sequences

- First we discuss a naive merging method for *merging* sorted sequences of length *m* and *n*. This algorithm is based on a *ranking method* and uses $T(m,n) = O(\log(n+m))$ and $W(m,n) = O((n+m)\log(n+m))$. This algorithm is **not** weakly optimal.

- Using a *limited ranking method* by a divide & conquer approach we will achieve a weakly optimal parallel algorithm for *merging* two sorted sequences using $T(m,n) = O(\log(n+m))$ and $W(m,n) = O(n+m)$.

- This parallel merging algorithm then will be used to achieve a weakly optimal *sorting algorithm* using $T(n) = O(\log^2 n)$.

Merging by ranking

- We reduce the merging problem to computing ranks of elements y w.r.t. a sequence X:

  $$\text{rank}(y : X) = |\{x \in X : x \leq y\}|$$

  = # elements in X *less than or equal to* y.

- Note that for *sorted* arrays A[1..n], we have

  $$\text{rank}(y : A[1..n]) = \max[\{i \mid A[i] \leq y, 1 \leq i \leq n\} \cup \{0\}]$$

  in such cases rank(y : A[1..n]) can be computed in O(log n) sequential time by applying *binary search.*

- As a special case for sorted sequences A (with unique elements) we have rank(A[i] : A) = i.

Merging by ranking

- We obtain the merge C of two *sorted* arrays A[1..m] and B[1..n] by using *ranking* for each element x in A and in B to obtain its position in C.

- We assume that both A and B consists of *unique* values.

- Since rank(A[i] : A) = i and rank(B[j] : B) = j we might compute C as follows:
  - for every i=1..m   C[i + rank(A[i] : B)] := A[i]
  - for every j=1..n   C[j + rank(B[j] : A)] := B[j]

- However, this is not completely correct:
  we did not take into account elements A[i] and B[j] such that A[i] = B[j].

  see the following example:

Example

- Let

  A = (2, 9 ,11, 13, 18, 23) and

  B = (3, 7, 9, 14, 17, 54)

  be *sorted integer* sequences.

  Let C be their merge

  C = (2, 3, 7, 9, 9, 11, 13, 14, 17, 18, 23, 54)

- Consider the element A[3] = 11.

  The index position of 11 in C is

  3 + rank(11:B) = 3 + 3 = 6.

  This element is unique.

- Consider A[2] = B[3] = 9.

  Now the index position of A[2] in C is

  2 + rank(9:B) = 2 + 3 = 5,

  and the index position of B[3] in C is:

  3 + rank(9:A) = 3 + 2 = 5

  but the index position of *the first* 9 in C is 4.

Example

In general, to achieve the *first* index position of a number A[i] that may occur twice in the merged sequence C we should compute its index in C as follows:

rank(A[i]:A) + rank(A[i]-1: B) = i + #elems in B less than A[i].

So, in the previous example we compute the index of the first 9 in C by

rank(9 : A)+rank(9-1 : B) = 4

In general, the position of the *i-th* element A[i] in A in the resulting merged array C is

index(A[i] : C) = i + rank(A[i]-1: B)

Analogously,

index(B[j] : C) = j + rank(B[j]: A)

First attempt: naive parallel merge

parmerge(A,B,n,m)

1. <u>for</u> i=1 <u>to</u> n <u>pardo</u>
   AA[i] := rank(A[i]-1,B);
   C[i+AA[i]]:=A[i];

the index position i+ AA[i] of A[i] in C equals i + #{ B[j] < A[i] }

2. <u>for</u> i=1 <u>to</u> m <u>pardo</u>
   BB[i] := rank(B[i],A);
   C[i+BB[i]]:=B[i];

the index position i+BB[i] of B[i] in C equals i + #{ A[j] ≤ B[i] }

3. <u>return</u> C

C is the sorted merge of A and B

A = 2 4 8   B = 1 5 7

AA = 1 1 3

BB = 0 2 2

C[1 + 1] = A[1] = 2

C[2 + 1] = A[2] = 4

C[3 + 3] = A[3] = 8

C[1 + 0] = B[1] = 1

C[2 + 2] = B[2] = 5

C[3 + 2] = B[3] = 7

C = 1 2 4 5 7 8

First attempt: naive parallel merge

This algorithm is not weakly optimal

parmerge(A,B,n,m)

1. for i=1 to n pardo
   AA[i] := rank(A[i]-1,B);
   C[i+AA[i]]:=A[i]

   od;

T = O(log m); W = n log m

the index position i+ AA[i] of A[i] in C equals i + #{ B[j] < A[i] }

2. for i=1 to m pardo
   BB[i] := rank(B[i],A);
   C[i+BB[i]]:=B[i];

   od;

T = O(log n); W = m log n

the index position i+BB[i] of B[i] in C equals i + #{ A[j] ≤ B[i] }

3. return C

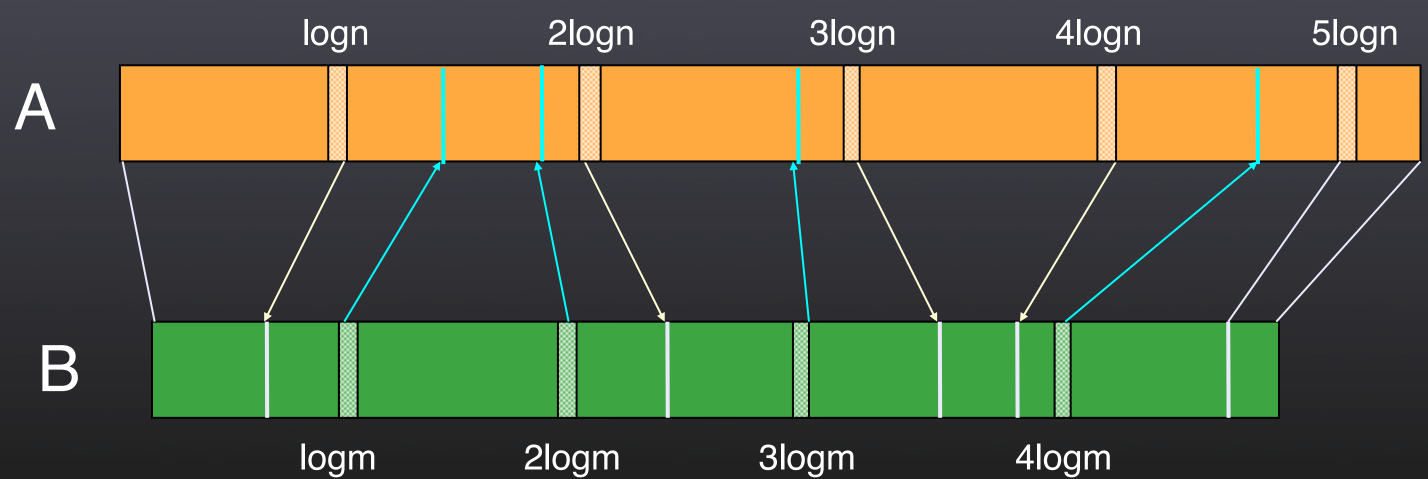
C is the sorted merge of A and B

T = O(log (n + m));
W = O((n + m) log (n+m))

improving naive parallel merge

Idea:

Given a sorted array A of n elements and an array B of m elements, find the index positions of every ⌈log n⌉-th element in A w.r.t. B and every ⌈log m⌉-th element in B w.r.t. A using the *ranking operation* in parallel.



improving naive parallel merge

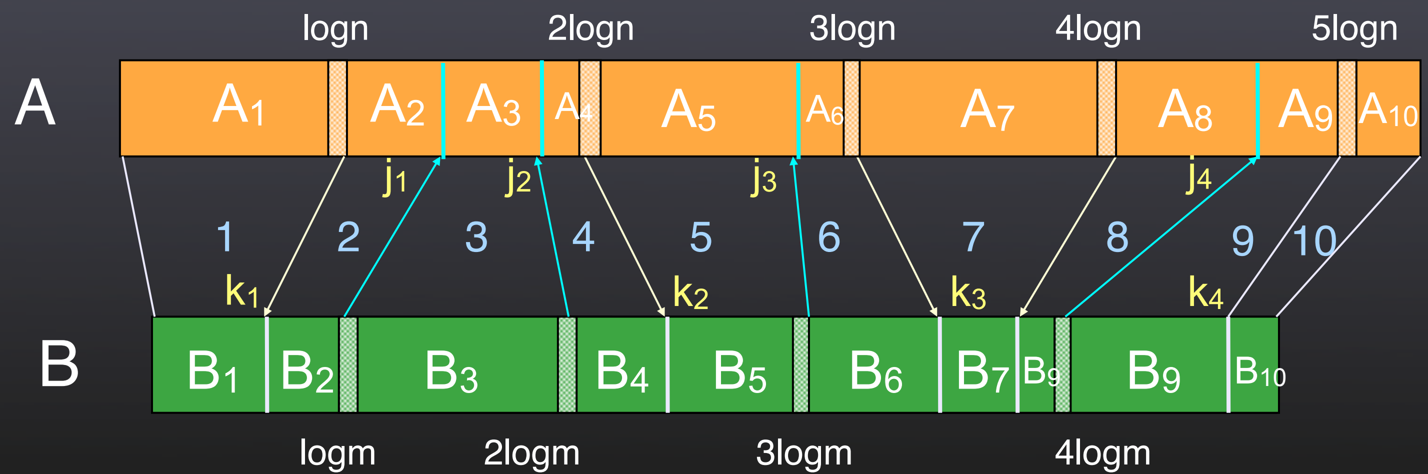
Observation 1

For every $x \in A_i \cup B_i$ and for every $y \in A_{i+k} \cup B_{i+k} : x \leq y$

Proof: Convince yourself!

Corollary

For every $i \neq j$, (A_i, B_i) and (A_j, B_j) can be merged independently and the resulting merged sequences $C_1, C_2, \ldots C_k$ can be simply concatenated.



observation: lines don't cross

improving naive parallel merge

Observation 2

Every block (A_i, B_i) consists of O(log n+m) elements

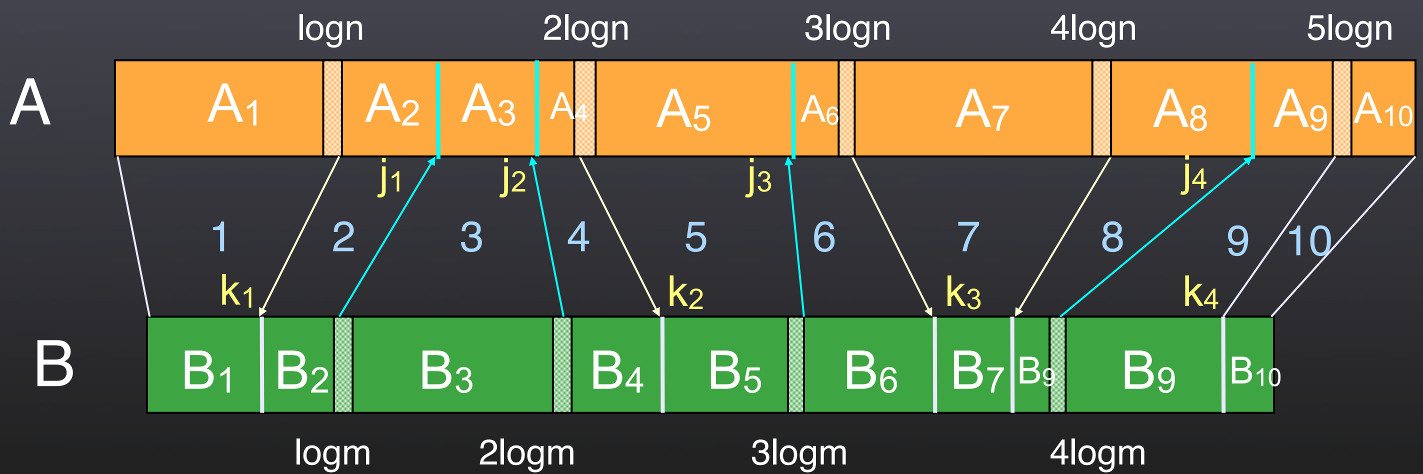
Proof:

Every block A_i consists of O(log n) elements, every block B_i consists of O(log m) elements.

In total they contain

$$O(\log n) + O(\log m) = O(\log n + \log m) = O(\log(n+m))$$

elements.



observation:
lines don't cross

improving naive parallel merge

Observation 2

In total there are $O((n+m)/\log(n+m))$ blocks.

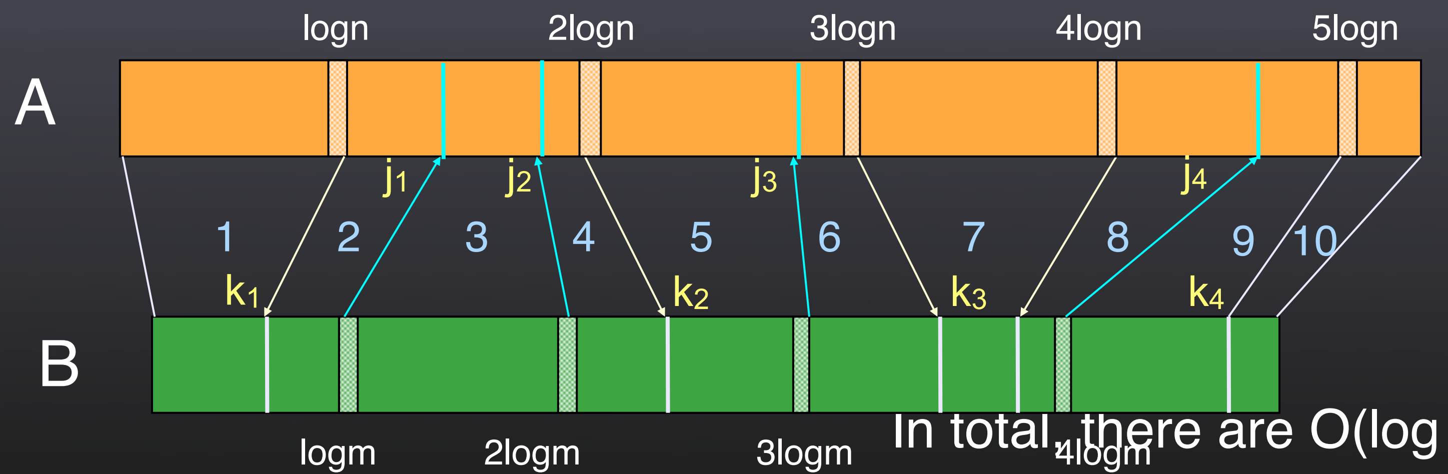
Proof:

There are $O(n/\log n + m/\log m)$ blocks.

If $2 \leq n \leq m$ then $O(n/\log n + m/\log m) \leq O(m/\log m) \leq O((m+n)/\log(n+m))$

else if $2 \leq n \leq m$ then $O(n/\log n + m/\log m) \leq O(n/\log n) \leq O((m+n)/\log(n+m))$

Hence, for $2 \leq n,m$ $O(n/\log n + m/\log m) \leq O((m+n)/\log(n+m))$.

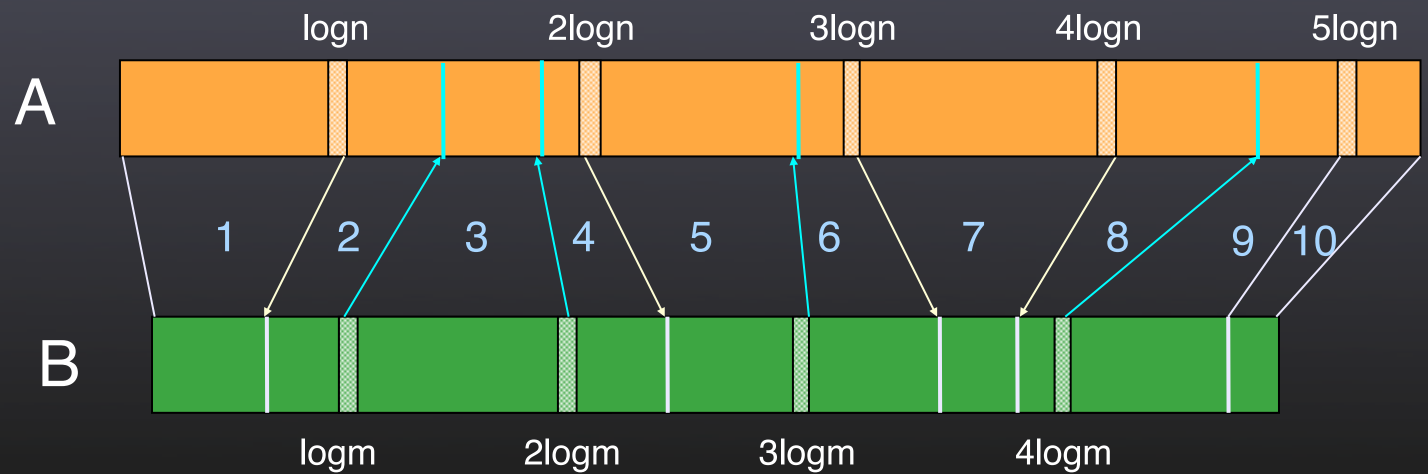


In total, there are O(log n+m) blocks.

improving naive parallel merge

So the total costs of the algorithm consist in

1. performing the ranking operations in parallel
2. merging the resulting blocks (Ai, Bi) in parallel
3. concatenating the results of these merging processes into a sorted sequence (by balanced tree)

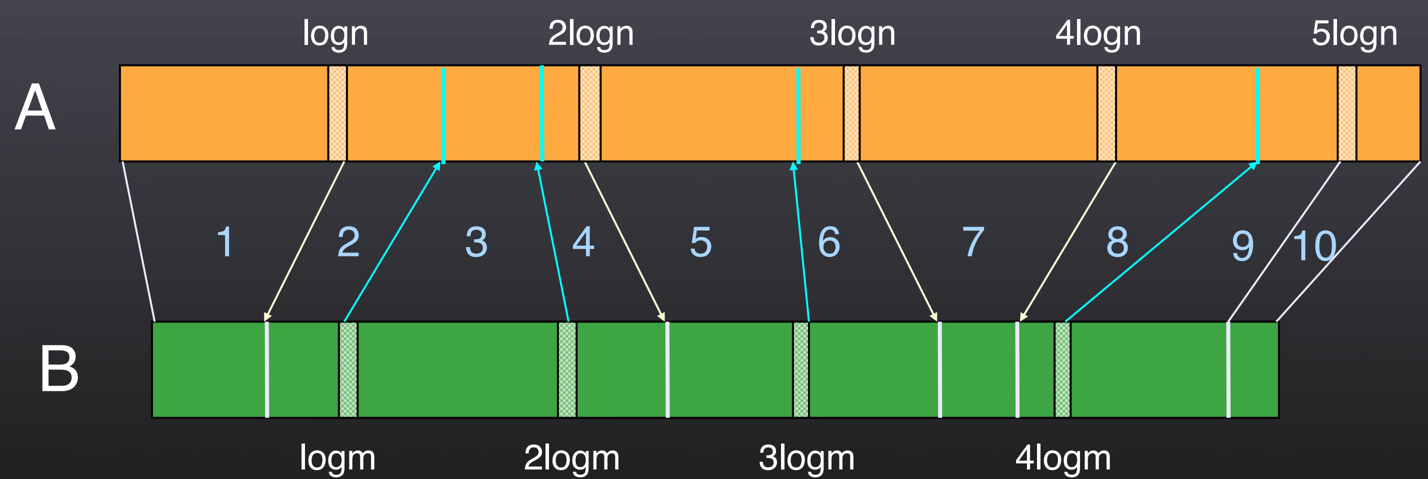


improving naive parallel merge

So the total costs of the algorithm consist in

1. performing the ranking operations in parallel — $T = O(\log (n+m)), W = O(n+m)$
2. merging the resulting blocks (Ai, Bi) in parallel — $T = O(\log (n+m)), W = O(n+m)$
3. concatenating the results of these merging processes into a sorted sequence (by balanced tree) — $T = O(\log ((n+m)/\log(n+m))), W = O(n+m)$

$T = O(\log (n+m)), W = O(n+m)$



Merging improved

improved_merge(A,B,n,m)

1. a:= log n; b := log m; AA[0]:=0;
2. for 1 ≤ i ≤ n/a, 1 ≤ j ≤ m/b pardo

   AA[i] := rank(A[i x a] - 1, B);

   BB[j] := rank[B[j x b], A);

3. for 1 ≤ i ≤ n/a, 1 ≤ j ≤ m/b pardo

   C[AA[i] + a x i]:= A[a x i],

   C[BB[i] + b x i]:= B[b x i],

4. for 0 ≤ i ≤ n/a, 1 ≤ j ≤ m/b pardo

   seqmerge(i.a , AA[i]),

   seqmerge(BB[j], j.b)

algorithm is weakly optimal

T = O(log (n+m)); W = O(n+m)

seqmerge(i,j) is a sequential merging procedure merging A[i+1, i+2, ...] with B[j+1, j+2, ...] into C[i+j+1, i+j+2, ...] until an element of C is encountered that has obtained a value in step 2

From merging to sorting

The previous algorithm can be easily adapted to obtain a weakly optimal *sorting* algorithm:

```
par_mergesort(X)
    begin
        if |X| =1 then return X
          else
            for 1 ≤ i ≤ 2 pardo
                X_i = par_mergesort([x_{1+n/2.(i-1)}, x_{2+n/2.(i-1)}, ... x_{n/2+n/2.(i-1)}]);
            improved_merge(X_1, X_2, n/2, n/2);
    end
```

T = O(1) ; W = O(1)

T = T(n/2) ; W = 2T(n/2)

T = O(log n) ; W = O(n)

$T(n) = T(n/2) + O(\log n) \Rightarrow T(n) = O(\log^2 n)$
$W(n) = 2W(n/2) + O(n) \Rightarrow W(n) = O(n \log n)$

Fast Simple Merge Sort

Theorem *fast-merge algorithm*: (without proof)

There exits a parallel algorithm to merge two sorted sequences of length n with

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

We use this algorithm to adapt the sequential merge-sort algorithm

- elements of the to be sorted array A are considered as *leaves* of a binary tree;
- in *$O(\log n)$*-iteration steps sorted subarrays of children of a node *x* are iteratively merged using *fast-merge* to a sorted array belonging to the node *x*.

Simple Merge Sort

simple_merge_sort(X)

input X[1 . . n], $n = 2^k$
output balanced bintree T with leaves X[i];
for every $0 \le h \le \log n$, L(h,j) is a sorted list of elements from the subtree with root (h,j)

begin

1. for $0 \le j \le n$ pardo
   L(0,j) := X[j];

   T = O(1), W = O(n)

2. for h = 1 to log n do
   for $1 \le j \le n/2^h$ pardo
   L(h,j) = *fastmerge* (L(h -1, 2j-1) , L(h -1, 2j))

   T = O(log n log log n),
   W = O(n log n)

3. return L(log n, 1)

end

T = O(log n log log n),
W = O(n log n)

Contents

Sorting methods

- sorting by merging
  - lowerbound for sorting
  - naive parallel sorting by merging
  - improved parallel sorting by merging
  - parallel sorting by fast merging
- enumeration sort
- random quicksort

enumeration sort (i)

To sort a sequence A[1..n] of elements into a sorted one,
it suffices to determine for each A[i] it *rank* rank[A[i], A] in A.
Determining the rank of an element in an unsorted sequence
A[1..n] will cost you O(n) time.

If we want to find a suitable parallel sorting technique based on this idea we have to concentrate on fast computation of the ranking procedure, since rankings of the individual elements can be done in parallel.

To compute the rank of each element, we present two algorithms

- one using a CRCW PRAM with *additive-write conflict resolution* i.e. if several processes write to the same location, the contents are *added* to that location (see Grama p. 415)
- the other by using a CRCW PRAM using a balanced tree computation that does not use additive write conflict resolution.

enumeration sort (i)

The algorithm below uses a CRCW PRAM with *additive-write conflict resolution* (see Grama p. 415)

enumerationsort(A,n)

1. for $1 \le i \le n$ pardo C[i] := 0
2. for $1 \le i, j \le n$ pardo
   if (A[i] < A[j]) or (A[i] = A[j] and i < j)
   then C[j] := 1
3. for $1 \le j \le n$ pardo
   A[C[j]+1] := A[j]

Due to additive resolution,

C[j] is now equal to the number of elements in A that are less than A[j] or equal to A[j] but have a smaller index. Hence, C[j] + 1 is the index of A[j] in the sorted variant of A.

enumeration sort (i)

The algorithm below uses a CRCW PRAM with *additive-write conflict resolution* (see Grama p. 415)

enumerationsort(A,n)

1. for $1 \leq i \leq n$ pardo $C[i] := 0$

2. for $1 \leq i, j \leq n$ pardo
   if $(A[i] < A[j])$ or $(A[i] = A[j]$ and $i < j)$
   then $C[j] := 1$

3. for $1 \leq j \leq n$ pardo
   $A[C[j]+1] := A[j]$

$T(n) = O(1), W(n) = O(n)$

$T(n) = O(1), W(n) = O(n^2)$

$T(n) = O(1), W(n) = O(n)$

$T(n) = O(1), W(n) = O(n^2)$

enumeration sort (ii)

We can easily remove the restriction of using a CRCW PRAM with additive write resolution by using a *balanced tree computation* scheme to compute the rank of the elements.
Instead of sorting in O(1)-time, this variant takes O(log n)-time, using $O(n^2)$ amount of work.

```
enumerationsort(A,n)
  for 1 ≤ i,j ≤ n pardo
     C[i,j] := 0
  for 1≤ i, j ≤ n pardo
     if (A[i] < A[j]) or (A[i] = A[j] and i < j)
        then C[i,j] := 1
  for 1 ≤ j ≤ n pardo
        B[j] := prefixsum(C[.,j],n)
  for 1 ≤ i ≤ n pardo
            A[1+B[i]] := A[i]
```

For each A[j] the total number of elems A[i] preceding A[j] is determined in parallel using a prefix computation scheme and the result is stored in B[j].

Time: O(log n)

enumeration sort (ii)

We can easily remove the restriction of using a CRCW PRAM with additive write resolution by using a *balanced tree computation* scheme to compute the rank of the elements.
Instead of sorting in O(1)-time, this variant takes O(log n)-time, using $O(n^2)$ amount of work.

total: $T(n) = O(\log n)$, $W(n) = O(n^2)$

```
enumerationsort(A,n)
  for 1 ≤ i,j ≤ n pardo
     C[i,j] := 0
  for 1≤ i, j ≤ n pardo
     if (A[i] < A[j]) or (A[i] = A[j] and i < j)
        then C[i,j] := 1
  for 1 ≤ j ≤ n pardo
        B[j] := prefixsum(C[.,j],n)
  for 1 ≤ i ≤ n pardo
           A[1+B[i]] := A[i]
```

- first loop: $T(n) = O(1)$, $W(n) = O(n^2)$
- second loop: $T(n) = O(1)$, $W(n) = O(n^2)$
- third loop: $T(n) = O(\log n)$, $W(n) = O(n^2)$
- fourth loop: $T(n) = O(1)$, $W(n) = O(n)$

Contents

Sorting methods

- sorting by merging
  - naive parallel sorting by merging
  - improved parallel sorting by merging
  - parallel sorting by fast merging
- enumeration sort
- random quicksort

RandomQuicksort : sequential

```
quicksort (A, q, r)
begin
    if q < r then
        begin
            p : = random([q..r])
            s : = q
            for i = q+1 to r do
                if A[i] ≤ p then
                    s:= s+1;
                    swap(A[s], A[i]);
            swap(A[q],A[s]);
            quicksort(A,q,s);
            quicksort(A, s+1,r);
        end
end
```

call with quicksort(A,1,n)

p: pivot

result:
all elements in A[1, . . ., s] are ≤ p and all elements in A[s+1, . . ., r] are > p

perform quicksort on the resulting subarrays (divide-and conquer)

quicksort parallel: naively

naive idea

apply divide and conquer to do quicksort(A,q, s) and quicksort(A, s+1,r) in parallel.

comment

this is a *bad* idea: since only 1 processor has to perform the first partition step we have:
$T(n) = O(n)$ and $W(n) = O(n^2)$

improvement

apply the *partition-step* in parallel.
We use an $O(\log n)$ algorithm to perform the partition step.

A simple parallel variant

- random_quicksort(A,n)

  1. if $n < 30$ then

     sort A using any sorting algorithm;

     exit;

  2. select random element a from A;
  3. for $1 \le i \le n$ pardo

     if $A[i] < a$ then $mark[i] := 0$

     if $A[i] \ge a$ then $mark[i] := 1$

  $T = O(\log n)$ 4. B := compact(A, mark[], a);

     k := position of first element equal to a in B;

  5. C1 := random_quicksort(B[1..k-1],k-1),

     C2 := random_quicksort(B[k+1..n],n-k),

  6. return C1++[a] ++C2

compactification is the process of placing all elements marked with 0 before all elements marked with 1; see next slide.

average time: $T = O(\log^2 n)$

compact

- Given
  an array A[1..n] and an array mark[1..n] with mark[i] $\in$ {0,1}.

  To compute:
  array B : all elems A[j] with mark[j] = 0 preceding
  the elems A[j] with mark[j] = 1.

- Solution (sketch)
  Let C = prefixsum(mark, n). Using C, we can determine the number of 0's in the array mark: z = n - C[n] is the number of 0's. For each A[i] with mark[i] = 1, its index position in B therefore is z + C[i].

  Let cmark[i] = 1 if mark[i] = 0 and cmark[i] = 0 if mark[i] = 1.
  Let CC = prefixsum(cmark, n). Using CC, we can find the positions of the remaining elements as follows:
  For each A[i] with mark[i]= 0, its index position in B is CC[i].

Example of compact

- Suppose we have

$$A = [3,4,2,6,8,5]$$
$$mark = [0,1,0,1,1,1]$$

- Computing the prefixsum of mark gives

$$C = [0,1,1,2,3,4]$$

- Note that there are 6 - C[6] = 2 zero’s. Hence, the index positions of the entries A[i] with mark[i] = 1 in B are 3, 4, 5 and 6.
- Computing CC gives

$$CC = [1,1,2,2,2,2]$$

- Hence the index positions of the entries A[i] with mark[i]=0 in B are 1 and 2.
- As a result, we have

$$B = [\ 3,\ 2,\ 4,\ 6,\ 8,\ 5\ ]$$