Parallel Merging

Advanced Algorithms & Data Structures

Lecture Theme 15

Prof. Dr. Th. Ottmann

Summer Semester 2006

Parallel merging through partitioning

The partitioning strategy consists of:

- Breaking up the given problem into many independent subproblems of equal size
- Solving the subproblems in parallel

This is similar to the divide-and-conquer strategy in sequential computing.

The Merging Problem

Given a set S with a relation $\leq$, S is linearly ordered, if for every pair $a,b \in S$ either $a \leq b$ or $b \leq a$.

The merging problem is the following:

Input: Two sorted arrays $A = (a_1, a_2,..., a_m)$ and $B = (b_1, b_2,..., b_n)$ whose elements are drawn from a linearly ordered set.

Output: A merged sorted sequence $C = (c_1, c_2,..., c_{m+n})$.

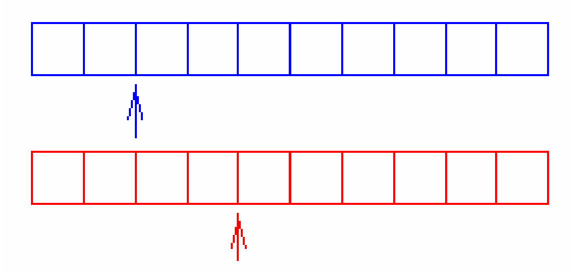
For example, if $A = (2,8,11,13,17,20)$ and $B = (3,6,10,15,16,73)$, the merged sequence

$C = (2,3,6,8,10,11,13,15,16,17,20,73)$.

Sequential Merging

A sequential algorithm

- Simultaneously move two pointers along the two arrays
- Write the items in sorted order in another array



- The complexity of the sequential algorithm is $O(m + n)$.
- We will use the partitioning strategy for solving this problem in parallel.

Partitioning and Merging

Definitions:

$rank(a_i : A)$ is the number of elements in A less than or equal to $a_i \in A$.

$rank(b_i : A)$ is the number of elements in A less than or equal to $b_i \in B$.

For example, consider the arrays:

$A = (2,8,11,13,17,20)$

$B = (3,6,10,15,16,73)$

$rank(11 : A) = 3$ and $rank(11 : B) = 3$.

Merging

- The position of an element $a_i \in A$ in the sorted array C is: $rank(a_i : A) + rank(a_i : B)$.

  For example, the position of 11 in the sorted array C is: $rank(11 : A) + rank(11 : B) = 3 + 3 = 6$.

Parallel Merging

- The idea is to decompose the overall merging problem into many smaller merging problems.
- When the problem size is sufficiently small, we will use the sequential algorithm.

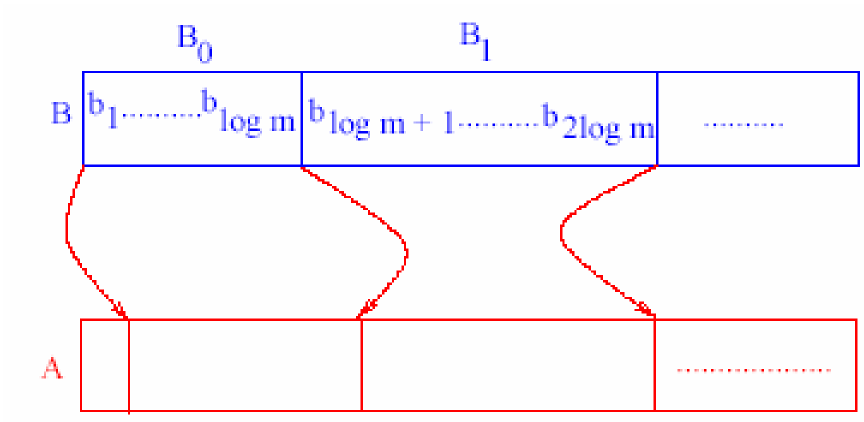
- The main task is to generate smaller merging problems such that:
  - Each sequence in such a smaller problem has $O(\log m)$ or $O(\log n)$ elements.
  - Then we can use the sequential algorithm since the time complexity will be $O(\log m + \log n)$.

Parallel Merging: Step1

Step 1. Divide the array *B* into blocks such that each block has log *m* elements. Hence there are *m*/log *m* blocks.

For each block, the last elements are

$i \log m$, $1 \le i \le m/\log m$



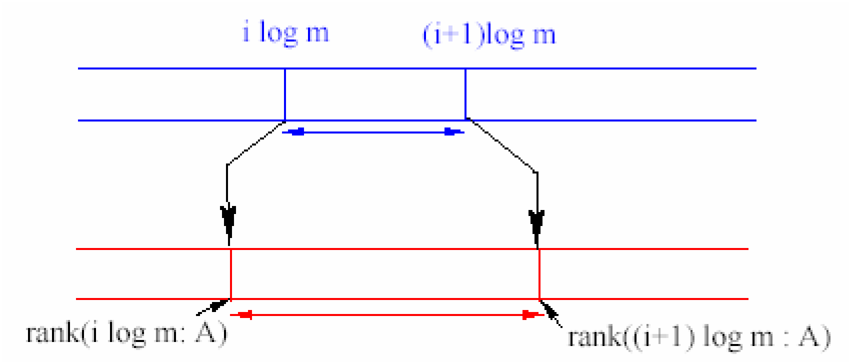
Parallel Merging: Step2

Step 2. We allocate one processor for each last element in *B*.

- For a last element $i \log m$, this processor does a binary search in the array *A* to determine two elements a_k, a_{k+1} such that $a_k \le i \log m \le a_{k+1}$.
- All the $m/\log m$ binary searches are done in parallel and take $O(\log m)$ time each.
- After the binary searches are over, the array *A* is divided into $m/\log m$ blocks.
- There is a one-to-one correspondence between the blocks in *A* and *B*. We call a pair of such blocks as matching blocks.

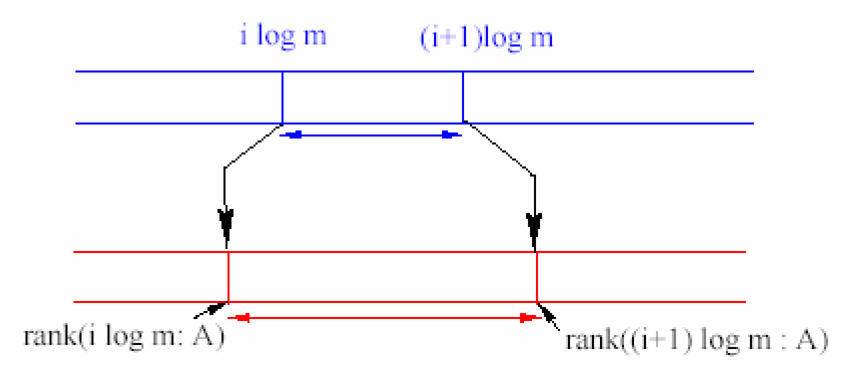
Parallel Merging: Matching Blocks

- Each block in A is determined in the following way.
- Consider the two elements $i \log m$ and $(i + 1) \log m$. These are the elements in the $(i + 1)$-th block of B.
- The two elements that determine $rank(i \log m : A)$ and $rank((i + 1) \log m : A)$ define the matching block in A

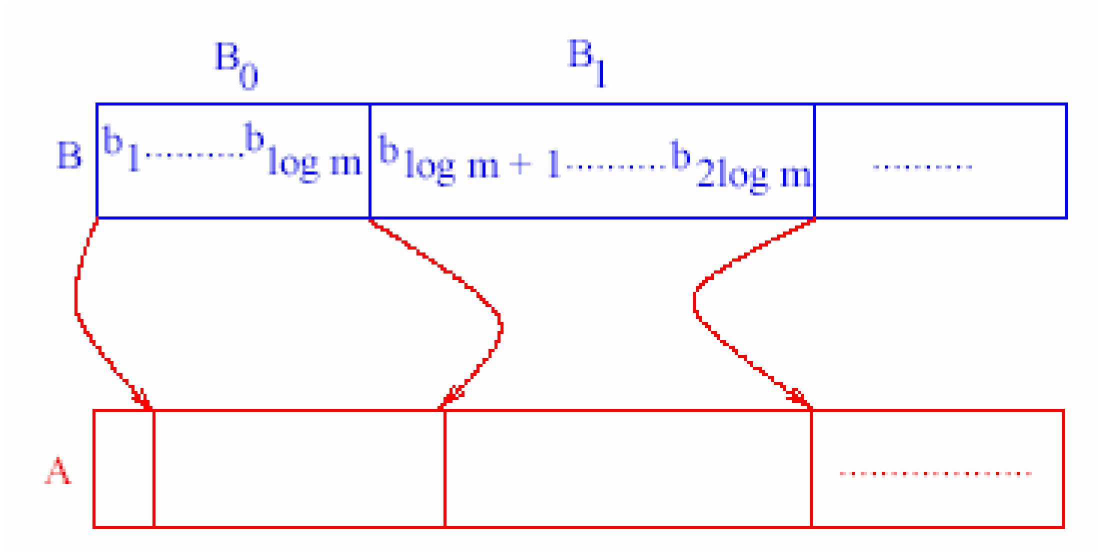


Reduction to smaller subproblems (1)

- These two matching blocks determine a smaller merging problem.
- Every element inside a matching block has to be ranked inside the other matching block.
- Hence, the problem of merging a pair of matching blocks is an independent subproblem which does not affect any other block.



Reduction to smaller subproblems (2)



- If the size of each block in *A* is *O*(log *m*), we can directly run the sequential algorithm on every pair of matching blocks from *A* and *B*.
- Some blocks in *A* may be larger than *O*(log *m*) and hence we have to do some more work to break them into smaller blocks.

Handling of large blocks

If a block in A_i is larger than $O(\log m)$ and the matching block of A_i is B_j, we do the following

- We divide A_i into blocks of size $O(\log m)$.
- Then we apply the same algorithm to rank the boundary elements of each block in A_i in B_j.
- Now each block in A is of size $O(\log m)$
- This takes $O(\log \log m)$ time.

Parallel Merging: Step 3

Step 3.

- We now take every pair of matching blocks from A and B and run the sequential merging algorithm.
- One processor is allocated for every matching pair and this processor merges the pair in $O(\log m)$ time.

We have to analyse the time and processor complexities of each of the steps to get the overall complexities.

Complexity analysis of Step 1

Complexity of Step 1

- The task in Step 1 is to partition B into blocks of size $\log m$.
- We allocate $m/\log m$ processors.
- Since B is an array, processor P_i, $1 \leq i \leq m/\log m$ can find the element $i * \log m$ in $O(1)$ time.

Complexity analysis of Step 2

Complexity of Step 2

- In Step 2, $m/\log m$ processors do binary search in array A in $O(\log n)$ time each.
- Hence the time complexity is $O(\log n)$ and the work done is $(m * \log n)/ \log m \le (m * \log(m + n)) / \log m \le (m + n)$ for $n, m \ge 4$. Hence the total work is $O(m + n)$.

Complexity analysis of Step 3

Complexity of Step 3

- In Step 3, we use $m/\log m$ processors
- Each processor merges a pair A_i, B_i in $O(\log m)$ time.Hence the total work done is m.

Theorem

Let A and B be two sorted sequences each of length n. A and B can be merged in $O(\log n)$ time using $O(n)$ operations in the CREW PRAM.