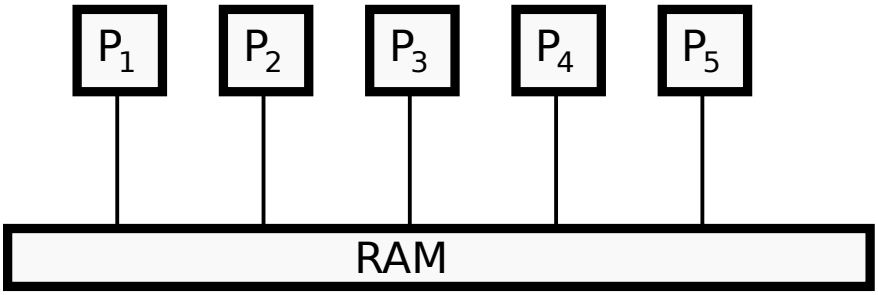
Algorithms for Memory Hierarchies - Lecture 10 - Parallel Algorithms

Lecturer: Nodari Sitchinava
Scribe: Mihai Herda

13.01.2013

1 The PRAM Model



The model that we shall use is the PRAM model, because of its similarities with modern multi-core CPUs. In this model, p processors share the RAM. Each processor can in one computation step:

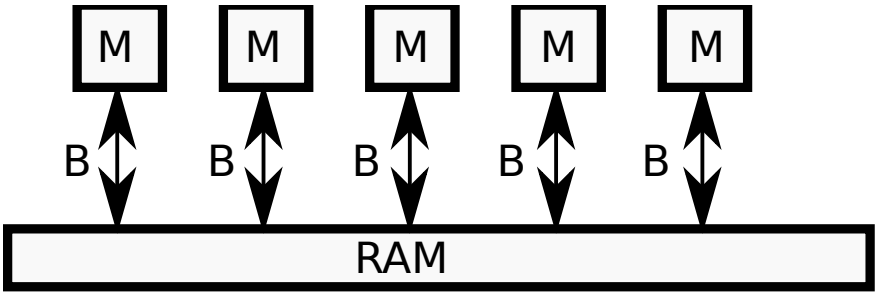
- Read one item
- Execute one operation
- Write a value

Sharing the RAM can cause certain problems that can be solved using locks. There are three models for the usage of locks:

- EREW - exclusive read exclusive write
- CREW - concurrent read exclusive write
- CRCW - concurrent read concurrent write

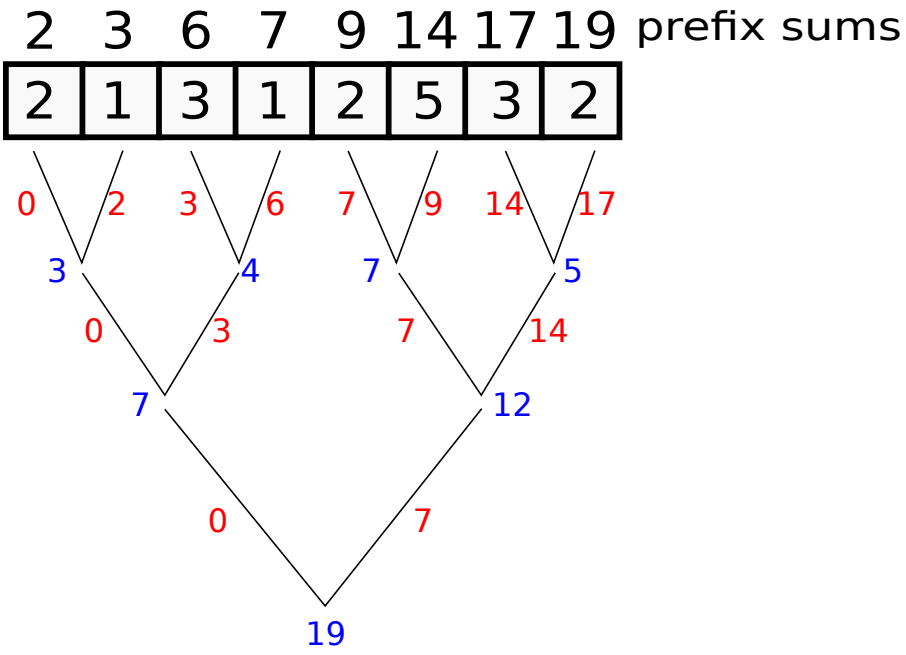
For our algorithms we will use the CREW model, meaning that we allow concurrent reads, but don't allow concurrent writes.

2 The Parallel External Memory Model



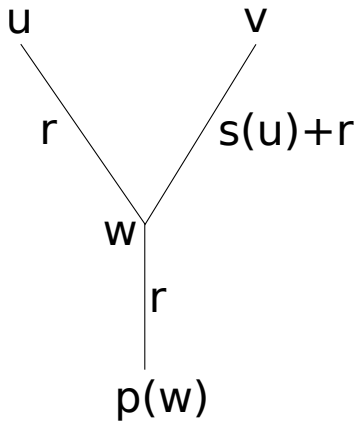
For the analysis of the IO complexity of our algorithms we shall use the parallel external memory model. Similarly to to pram model we have p processors that have common RAM. Additionally each processor has its own cache memory. Data is transferred between the cache and the RAM in blocks of size B. The cache of a processor is of size M. The IO complexity is given by the maximum IO Complexity for one of the processors.

3 Prefix Sums



The parallel computation of prefix sums takes places in two steps. In the first step we build a binary tree, where each node v contains the sum $s(v)$ of the values of its children. In the second step we do the following:

- pass the value r from the parent to the left child
- pass the value $r + s(leftchild)$ to right child
- for the root r is 0



The value of r that is passed to a node v represents the summation of all elements at the left of v. If we have n processors at our disposal we can compute the prefix sums for n elements in $O(\log n)$ time(the height of the binary tree).

3.1 Less than n processors

The previous value for the time complexity is valid only if we have a free processor for each computation at any time. But what happens when we have fewer processors? The standard strategy, when having fewer processors than optimal, is to simulate with one processor the work done by more than one processors in the optimal case. If we wish to compute the prefix sums with this algorithm using only one processor, then that processor must simulate the work of n processors. Because each one of the n processors required $O(\log n)$ time, the work done by a the single processor will take $O(n \log n)$ time. The sequential algorithm would need only $O(n)$ time. If a parallel algorithm executed by a single processor fares worse than the best sequential algorithm, then we say that the parallel algorithm is not work-optimal. If we wish to use p processors to compute the prefix sums with our algorithm, then the time needed would be $O(\frac{n}{p} \log n)$.

3.2 An improved algorithm

We can improve the parallel prefix sums algorithm in order to make it work-optimal. We divide the array in p sub-arrays of size $\frac{n}{p}$. We compute for each sub-array the prefix sums using the sequential algorithm. The binary tree will use this results and its leafs will no longer be the individual elements, but the sub-arrays. The time for the new algorithm consists of $O(\frac{n}{p})$, the time needed by the sequential algorithm for one sub-array and $O(\log p)$, the time needed by the parallel algorithm to work on the new binary tree of height p, resulting in a total time of $O(\frac{n}{p} + \log p)$. The IO complexity of the improved algorithm has the following components:

- $scan(\frac{n}{p}) = O(\frac{n}{pB})$ for the sequential algorithm
- $O(\log p)$ for constructing the binary tree

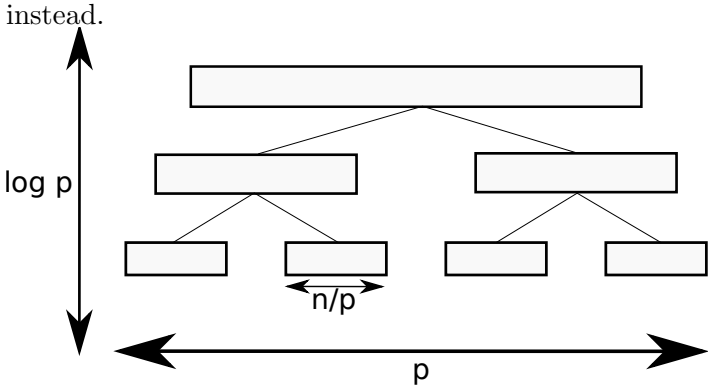
The total IO complexity is:

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

4 Merge Sort

```
msort(A[1 ... n])
B_1 = msort(A[1 ... n/2]);
B_2 = msort(A[n/2 + 1 ... n]);
return merge(B_1,B_2);
```

The most obvious step for transforming merge sort into a parallel algorithm is to run the sorting of the two sub-arrays on different processor. We shall use $\frac{p}{2}$ processors for each of the two sub-arrays. If we only have one processor at our disposal, we shall use the sequential merge sort instead.



The time needed for this parallel merge sort algorithm is:

$$\begin{aligned} T(n,p) &= T\left(\frac{n}{2},\frac{p}{2}\right) + \underbrace{O(n)}_{\text{merge}} \\ &= O\left(\frac{n}{p}\log\frac{n}{p} + n\right) \end{aligned}$$

If $p < \log n$:

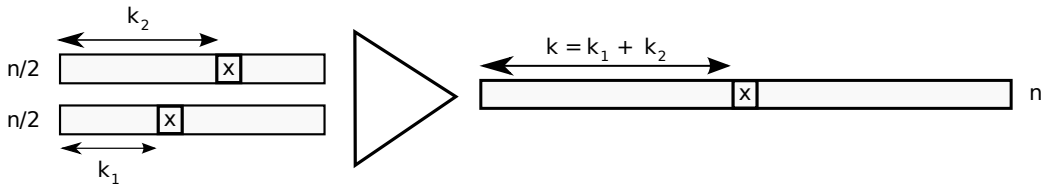
$$T(n,\log n) = O(n)$$

and the total run time is:

$$O\left(\frac{n\log n}{p}\right)$$

If p is too large, the merge step done by only one processor will become inefficient. We have to come up with a parallel merge algorithm.

4.1 Parallel Merge



Because the two sub-arrays that must be merged are sorted we can find out for each element x the number k of elements that are smaller than x. We do a binary search on each sub-array and need $O(\log n)$ time. This binary searches can be done in parallel, because we only need to read data. We then know where to write x in the merged array. With n processors the parallel merge needs $O(\log n)$ time. The time for parallel merge sort with parallele merge using n processors is:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + O(\log n) \\ &= \log 1 + \log 2 + \log 4 + \ldots + log\frac{n}{2} + logn \\ &= O(\log^2 n) \end{aligned}$$

If $p < \frac{n}{\log n}$ then the time is:

$$O\left(\frac{n \log n}{p}\right)$$

Because it makes use of binary sort while merging this algorithm is not IO efficient, because it jumps all over the place. We need to consider an algorithm that doesn't use binary search. As a note, the fastest known parallel merge sort algorithm is Cole's merge sort, which requires only $O(\log n)$ time for $p = n$.

5 Distribution Sort

Let's remember how EM distribution sort works:

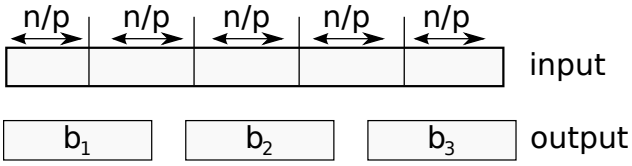
Distribution Sort
1. select $\sqrt{\frac{M}{B}}$ "good"pivots;
2. partition input into $\sqrt{\frac{M}{B}} + 1$ buckets;
3. recurse on each bucket;

The IO complexity of EM distribution sort is:

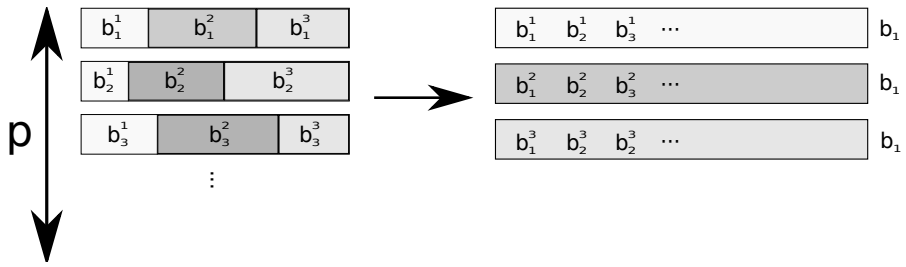
$$O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$$

The PEM distribution sort algorithm divides the input array in p sub-arrays of size $\frac{n}{p}$ each. These sub-arrays will then be sorted using EM distribution sort. We will now use parallel algorithms for steps 1 and 2.

5.1 PEM partition



We join the buckets of the p sub-arrays:



In order to know at what position we should copy a small bucket into a large bucket we need to compute the prefix sums for the d small buckets. If $d = B$ the IO complexity for calculating the prefix sums is:

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

For any value of d the IO complexity of computing the prefix sums is:

$$O\left(\frac{d}{B}\log p\right)$$

The total complexity of partitioning is:

$$O\left(\underbrace{\frac{n}{p\log p}}_{\text{scanning}} + \underbrace{\frac{d}{B}\log p}_{\text{prefix sums}}\right)$$

If $d < \dfrac{n}{p\log p}$ the IO complexity is $O\left(\dfrac{n}{pB}\right)$.