Algorithms for Memory Hierarchies
Lecture 11

Lecturer: Nodari Sitchinava
Scribe: Fabian Klute, Michael Hamann

1 PEM Distribution Sort

Last lecture we saw how partitioning can be parallelized in the PEM model. In this lecture we will concentrate on the pivot selection.

1.1 Median selection in parallel

The big picture of sequential median selection for an input of size N consists of the following steps:

1. Find medians in groups of 5
2. Find medians of these $N/5$ medians (recursively)
3. Partition the input around the median
4. Recurse on the appropriate partition

For the parallelization of the first step we can simply use groups of N/P elements and calculate the median of these P groups in parallel using the sequential median selection algorithm select($A^i[1, \dots, N/P]$, $N/2P$).

In the second step we need to select the median of P elements. Instead of a recursion we use PRAM merge sort using P processors in order to sort the P medians in $O(\log P)$ time.

For the third step we can use the parallel partition algorithm from the previous section. In the last step the recursion is executed with all processors.

For the running time of this algorithm we get the following recurrence:

$$T(n,p) = \underbrace{O\left(\frac{n}{p}\right)}_{\text{Step 1}} + \underbrace{O\left(\log p\right)}_{\text{Step 2}} + \underbrace{O\left(\frac{n}{p} + \log p\right)}_{\text{Step 3}} + \underbrace{T\left(\frac{3}{4}n, p\right)}_{\text{Step 4}}$$

For the base case of $n < N/P$ elements only one processor is used which leads to a recursion depth of $\log P$. This recurrence can be resolved to $O(n/p + \log^2 p)$.

For the I/O-complexity we get the following recurrence:

$$Q(n,p) = O\left(\frac{n}{pB}\right) + O\left(\log p\right) + O\left(\frac{n}{pB} + \frac{\log p}{B}\right) + Q\left(\frac{3}{4}n, p\right)$$

This leads to $Q(N,P) = O\left(\frac{N}{PB} + \log^2 p\right)$. Note that if $p < N/(B \log^2 N)$ there are just $O(\frac{N}{PB})$ I/O's.

1.2 Finding all pivots

For the distribution sort algorithm we do not only need to determine one pivot, but $d := \sqrt{M/B}$ pivots.

Instead of using the pivot selection algorithm d times we had the following two steps in the sequential algorithm:

1. Pick every dth element from each run for a total of
$$\frac{N}{M} \cdot \frac{M}{d} = \frac{N}{d}$$
elements.

2. Run the selection algorithm on N/d elements d times. Each run took $O(N/(dB)$ I/O's which results in $O(N/(dB))d = O(N/B)$ total I/O's for this step.

The first step can be trivially parallelized by grouping the blocks of M elements into P groups of $N/(P \cdot M)$ blocks and executing the first step for the different groups in parallel.

The simple solution for the second step is to execute the parallel selection algorithm on N/d elements d times. This gives us $O(N/(PB) + d \log P)$ I/O's. However we would like to get rid of the d factor in front of the $\log P$ term.

Thus instead of this simple solution we search the d pivots in parallel. In order to be able to execute multiple pivot searches in parallel we need to create d copies of the N/d elements so each of these copies can be partitioned independently. Then we use $p' := P/d$ processors to find each of the d pivots independently in parallel. This leads to $O(N/d \cdot 1/(p' \cdot B) + \log^2 p') = O(N/(PB) + \log^2 P/d)$ I/O's.

1.3 I/O complexity of PEM distribution sort

The last step of the PEM distribution sort algorithm, the recursion, is executed in parallel as long as there is more than one processor, otherwise the sequential EM distribution sort algorithm is used. The three steps (pivot selection, partitioning and recursion) lead to the following recurrence for I/O's:

$$Q(n,p) = \begin{cases} O\left(\frac{n}{pB} + \frac{\sqrt{M/B}}{B}\log p\right) + O\left(\frac{n}{pB} + \log^2 \frac{p}{\sqrt{M/B}}\right) + Q\left(\frac{n}{\sqrt{M/B}}, \frac{p}{\sqrt{M/B}}\right) & \text{if } p > 1 \\ O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right) & \text{if } p = 1 \end{cases}$$

If P is smaller than $\frac{N}{B\log^2 N}$ this recurrence can be resolved to

$$\begin{aligned} Q(N,P) &= \frac{N}{PB} + \underbrace{\frac{N/\sqrt{M/B}}{P/\sqrt{M/B}B} + \frac{N/\sqrt{M/B}^2}{P/\sqrt{M/B}^2 B} + \ldots +}_{\log_{\sqrt{M/B}} P \text{ terms}} \frac{N/P}{B}\log_{M/B}\frac{N/P}{B} \\ &= \frac{N}{PB}\log_{\sqrt{M/B}} P + 2\frac{N}{PB}\log_{M/B}\frac{N}{PB} \\ &= O\left(\frac{N}{PB}\log_{M/B}\frac{N}{B}\right) \end{aligned}$$

There is another more complicated PEM merge sort algorithm that achieves $O(\frac{N}{PB}\log_{M/B}\frac{N}{B})$ if $p < \frac{N}{B^2}$.

2 PEM Graph algorithms

In this section we look at Graph algorithms in the PEM model. As in the sequential case we start with list ranking. First in the PRAM, then in the PEM model.

2.1 List Ranking

List ranking is a basic problem in graph algorithms. We want to compute the rank of a node in a list, where the nodes are connected sequentially.

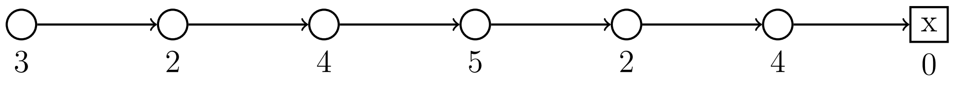

Figure 1: Example of a list ranking instance.

The example above shows a weighted list ranking instance, the weights are given below the nodes and the last node is used as a sentinel. The idea is to reduce list ranking to a prefix sum problem. In the above example the weights are chosen randomly, but if we set them all to one we can obviously compute the rank as a prefix sum. Doing this sequentially and without the notion of I/O's is easy, but if we take multiple processors and later I/O's into account, the problem becomes a bit more difficult to solve.

2.2 List Ranking in PRAM

If we don't care about I/O efficiency, than there is an easy list ranking algorithm known as "Wyllie's pointer hopping". The pseudocode of the algorithm is rather short and easy to understand:

Algorithm 1 Wyllie's pointer hoping algorithm to calculate list ranking.

```
for $i < \log N$ do
    on processor $p_i$ do:
    $weight(v) += weight(succ(v))$
    $succ(v) \leftarrow succ(succ(v))$
end for
```

Compute for each node the sum of its own weight and the weight of its successor, then the successor of the node is set to the successor of its successor. When the successor is the sentinel element nothing happens. This is repeated $\log N$ times, then all pointers point to the sentinel element and have calculated the prefix sums in reverse order. The operations are independent from each other and we can assign each node to a processor if the number of processor equals the number of nodes. The running time of "Wyllie's pointer hopping" is in $O(\text{sort}_p(N))$.

2.3 List Ranking in PEM

Obviously the above algorithm is not really good in terms of I/O's. In an earlier lecture we already saw a solution for the sequental case. The basic idea was to calculate an independent set of nodes and bridge them out. Then compute the prefix sum using a pairwise sum function

between nodes in the independent set and nodes outside of the set. When the sums are computed, we recursively calculate the prefix sum of the remaining nodes.
Now with parallel computing we have to adjust the way of finding the independent set. We opted for a coloring algorithm in the above solution. This approach won't work here, instead we look at a randomized coin toss algorithm to get an idea. For each node we flip a coin if it shows heads we take the node into the set, if it says tails we don't. To make sure we have an independent set, each node looks at its successor. If the successor flipped heads too, the node will not be in the independent set. It is provable, that the expected size of the set is $\frac{N}{4}$. This is good, but we aim for a deterministic solution. The following algorithm is called "deterministic coin toss".

2.3.1 Deterministic coin toss

Assume every node has an unique ID. We define a "ruler" as a node, which has the minimal ID among its neighbours (obviously we need forward and backward edges to do this). Now we look at the binary representation of a node ID and at the ID of its successor. We take the first digit where both IDs differ and remember, if it is an one or a zero. With this information each node calculates a tag with $tag(v) = 2i + b$, where i is the position of the difference and b is zeor or one. It's easy to see that the tag of a node v and its successor are different. Either the index differs or the, if i is the same b has to differ, else the IDs are not unique
Now we can pick "rulers" by the tag. The chosen nodes form an independent set, since we build the tags in a way, where no successor of a node has the same tag as his predecessor. The remaining question is, how big is this set? The distance of two nodes in the independent set is at most $2 * \#$distinct tag values. Each ID can be represented with $\log N$ bits and each tag is smaller or equal $2(\log N - 1) + 1 = 2\log N - 1$.You can see that the distance between two rulers is in $O(\log N)$. The size of our set is at least $\Omega(\frac{N}{\log N})$. A simple example of this procedure is shown in figure 2.

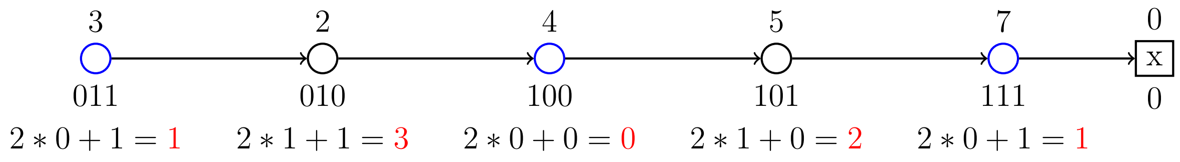

Figure 2: Calculating the tags out of IDs. The red marked numbers are the resulting tags and the blue nodes would be in an independent set.

Now we can make this set even bigger, by simply computing tags of tags. Tags of tags only need $\log\log N$ bits and following the above argumentation we get an independent set of size $\Omega(\frac{N}{\log\log N})$. Repeating this k-times we get a nearly constant function, where k is $\log^* N$. Using this independent set to calculate list ranking in PEM, we get $O(sort_p \log^* N)$ I/O's.