PRAM Sorting Algorithms

Sorting is probably the single most studied topic in the world of algorithm design (whether sequential or parallel). Its popularity stems from its relative simplicity (in the sense that the problem itself is simply and clearly defined, without being completely trivial) and its wide applicability. Consequently, we will investigate a range of proposed parallel sorting algorithms. This note contains a discussion of two PRAM sorting algorithms.

We have already seen one PRAM sorting algorithm, which ran in constant time on n^2 processors. This algorithm was based on a simple sequential algorithm called **enumeration sort** which has $\Theta\left(n^2\right)$ run time. Thus the PRAM version was an efficient implementation of a rather inefficient sequential algorithm and was therefore not cost optimal. Many better sequential algorithms exist, with the two best known being **mergesort** and **quicksort**. Mergesort has $\Theta\left(n \log n\right)$ run time, while quicksort performs similarly on average, but with $\Theta\left(n^2\right)$ run time in the worst case. However, the introduction of a little randomisation to pivot selection makes quicksort a competitive algorithm (with small constants hidden in the asymptotic notation and sorting **in place**), popular in practice. Neither PRAM algorithm we present are strictly cost-optimal. The CREW version of mergesort is a a factor of $\log n$ out, while the CRCW variant of quicksort is 'cost-optimal on average', in a similarly practical way to sequential quicksort (in as much as any CRCW PRAM algorithm is practical, which is a separate debate of course!).

CREW Mergesort

Mergesort is a divide-and-conquer algorithm in which all the work is done in the 'combine' phase. This involves merging two independently sorted sequences of the same length. To avoid a $\Theta\left(n\right)$ time bottleneck in the 'root' process, we must devise a scheme for performing this task in parallel, with a sensible number of processors. Our approach borrows something from

enumeration sort, in that we will assign one processor to each item. The processor will have the job of working out its items position (or **rank**) in the merged sequence. Once all positions are known, the items are moved to their correct position in a single parallel step. We will assume that all the items are distinct (but the algorithm can be tweaked to relax this condition - can you think how?).

We exploit the fact that the two sequences are already sorted to simplify the task of calculating rank. Firstly, we note that the final rank is the sum of an item's rank in its own sequence (which we know already) and what would be its rank if inserted into the other sequence. This can be calculated in $O(\log s)$ time by a single processor, for a sequence of length s, using a standard binary search (see any standard sequential algorithms text if this is unfamiliar). Since s such searches will be proceeding concurrently, we require a concurrent read PRAM.

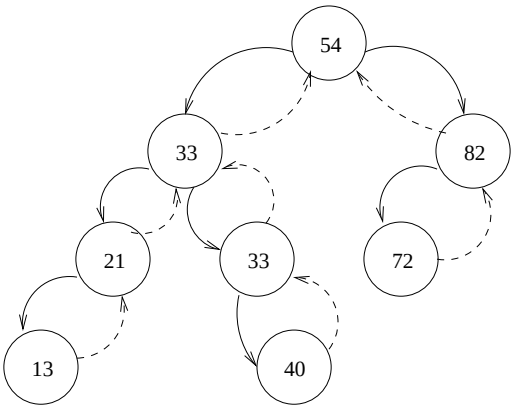
Now consider the big picture with one processor per item throughout. The merging process begins with $\frac{n}{2}$ pairs of sequences of length 1 being merged in parallel concurrently (i.e. with nested parallelism) then $\frac{n}{4}$ pairs of sequences of length 2, and so on, until the final step which merges 2 sequences of length $\frac{n}{2}$. Each such step (in a sequence of $\Theta(\log n)$ steps) takes time $\Theta(\log s)$ with a sequence length of s. Summing these, we obtain an overall run time of $\Theta\left(\log^2 n\right)$, for n processors, which is not quite (but almost!) cost-optimal.

CRCW Quicksort

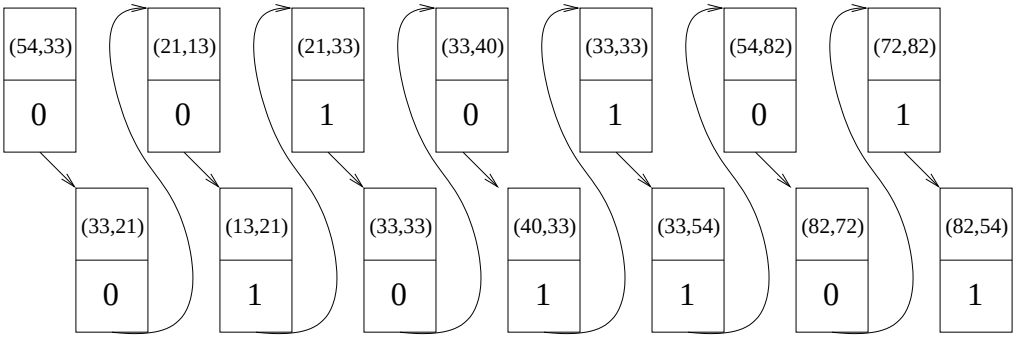
In contrast to mergesort, sequential quicksort is a divide & conquer algorithm in which all the work is in the divide phase. A **pivot** value is chosen from the sequence and the values are re-arranged so that all which are smaller than (or equal to) the pivot come before it, and all those larger come after it. Our CRCW quicksort notes that this amounts to computing a tree of pivots, from which ranks can be computed, before rearranging the data, as in CREW mergesort. The tree construction is described in detail in Kumar and cleverly exploits concurrent write with arbitrary clash resolution to randomize the choice of pivot. The computation of ranks is not covered in detail and so we now elaborate on this.

We first note that if we knew how many nodes were in each sub-tree then we could easily compute the ranks with a simple parallel sweep down the tree. The rank of the root is simply the number of nodes in its left sub-tree. The rank any other node depends upon whether the node is itself a left or a right child. For left children, rank is the parent's rank minus the size of the node's right sub-tree (because these are all larger the node) minus 1 (for the node itself). Similarly, for right children rank is the parent's rank plus the size of the left tree plus 1. Try it with a few examples! Since the sweep proceeds level by level down the tree, but in parallel at each level, it will take time proportional to the depth of the tree, which is expected (because of the randomization in the tree creation phase) to be $\Theta(logn)$ time.

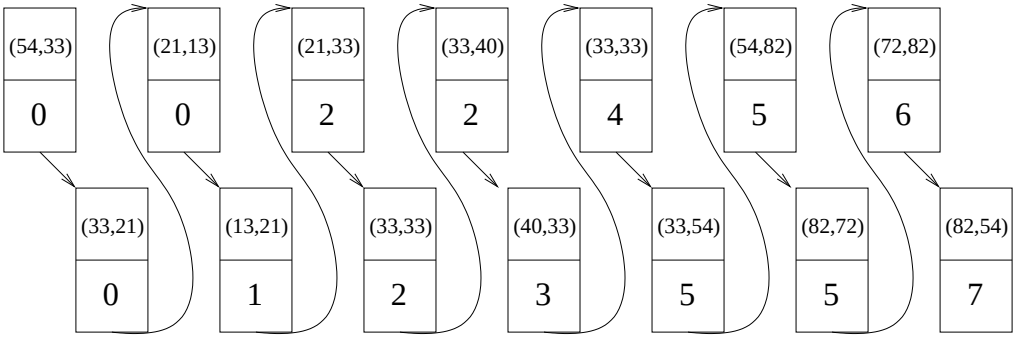
It now remains to work out how to compute the sub-tree sizes quickly. Given the tree of pivots, we think of each tree edge as consisting of two edges, one going down (towards the leaves) and one coming back up (towards the root). The size of a sub-tree is the number of up edges it contains (again, try a few examples to convince yourself of this). We label each such edge with an extra integer, 0 for "down" edges and 1 for "up" edges.



Now consider the order in which these edges would be visited by a "depth-first" traversal of the tree (in other words a tour of the tree starting at the root, and recursively visiting the left sub-tree of a node, then recursively visiting its right sub tree). This gives us a list of edges



We want to know, for each node in the original tree, how many up edges are visited between the first and last time that node appears in the edge list. A little thought reveals that this is related to a prefix (with addition) of the 0/1 values with which we labelled the list.



For any node in the original tree, the relevant number of up edges (in other words the size of the sub-tree) is computed by subtracting the prefix count as it stood when the node was first entered, from the count when it was finally left. For example the higher of the two nodes containing 33 in this example roots a subtree of size 5 which is $5 - 0$, while the node containing 82 roots a subtree of size 2 which is $7 - 5$. The root is an easy special case, having a count one more than the final value in the prefix. Of course, our data structures assume that we know the number of nodes in the whole tree anyway.