4

List Ranking

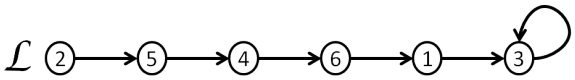
"Pointers are dangerous in disks!"

This lecture attacks a simple problem over lists, the basic data structure underlying the design of many algorithms which manage interconnected items. We start with an easy to state, but inefficient solution derived from the optimal one known for the RAM model; and then discuss more and more sophisticated solutions that are elegant, efficient/optimal but still simple enough to be coded with few lines. The treatment of this problem will allow also us to highlight a subtle relation between *parallel computation* and *external-memory computation*, which can be deployed to derive efficient disk-aware algorithms from efficient parallel algorithms.

Problem. *Given a (mono-directional) list $\mathcal{L}$ of n items, the goal is to compute the distance of each of those items from the tail of $\mathcal{L}$.*

Items are represented via their ids, which are integers from 1 to n. The list is encoded by means of an array $\mathtt{Succ}[1,n]$ which stores in entry $\mathtt{Succ}[i]$ the id j if item i *points to* item j. If t is the id of the tail of the list $\mathcal{L}$, then we have $\mathtt{Succ}[t] = t$, and thus the link outgoing from t forms a *self-loop*. The following picture exemplifies these ideas by showing a graphical representation of a list (left), its encoding via the array Succ (right), and the output required by the list-ranking problem, hereafter encoded in the array $\mathtt{Rank}[1,n]$.

This problem can be solved easily in the RAM model by exploiting the constant-time access to its internal memory. We can foresee three solutions. The first one scans the list from its head and computes the number n of its items, then re-scans the list by assigning to its head the rank $n-1$ and to



id	Succ	Rank
1	3	1
2	5	5
3	3	0
4	6	3
5	4	4
6	1	2

FIGURE 4.1: An example of input and output for the List Ranking problem.

every subsequent element in the list a value decremented by one at every step. The second solution computes the array of predecessors as $\texttt{Pred}[\texttt{Succ}[i]] = i$; and then scans the list backward, starting from its tail t, setting $\texttt{Rank}[t] = 0$, and then incrementing the Rank's value of each item as the percolated distance from t. The third way to solve the problem is *recursively*, without needing (an explicit) additional working space, by defining the function $\texttt{ListRank}(i)$ which works as follows: $\texttt{Rank}[i] = 0$ if $\texttt{Succ}[i] = i$ (and hence $i = t$), else it sets $\texttt{Rank}[i] = \texttt{ListRank}(\texttt{Succ}[i]) + 1$; at the end the function returns the value $\texttt{Rank}[i]$. The time complexity of both algorithms is $O(n)$, and obviously it is optimal since all list's items must be visited to set their n Rank's values.

If we execute this algorithm over a list stored on disk (via its array Succ), then it could elicit $\Theta(n)$ I/Os because of the arbitrary distribution of links which might induce an irregular pattern of disk accesses to the entries of arrays Rank and Succ. This I/O-cost is significantly far from the lower-bound $\Omega(n/B)$ which can be derived by the same argument we used above for the RAM model. Although this lower-bound seems very low, we will come in this lecture very close to it by introducing a bunch of sophisticated techniques that are general enough to find applications in many other, apparently dissimilar, contexts.

The moral of this lecture is that, in order to achieve I/O-efficiency on *linked* data structures, you need to avoid the *percolation* of pointers as much as possible; and possibly dig into the wide parallel-algorithms literature (see e.g. [2]) because efficient parallelism can be turned surprisingly into I/O-efficiency.

4.1 The pointer-jumping technique

There exists a well-known technique to solve the list-ranking problem in the parallel setting, based on the so called *pointer jumping* technique. The algorithmic idea is pretty much simple, it takes n processors, each dealing with one item of $\mathcal{L}$. Processor i initializes $\texttt{Rank}[i] = 0$ if $i = t$, otherwise it sets $\texttt{Rank}[i] = 1$. Then executes the following two instructions: $\texttt{Rank}[i] \mathrel{+}= \texttt{Rank}[\texttt{Succ}[i]]$, $\texttt{Succ}[i] = \texttt{Succ}[\texttt{Succ}[i]]$. This update actually maintains the following invariant: Rank[i] *measures the distance (number of items) between i and the current* Succ[i] *in the original list*. We skip the formal proof that can be derived by induction, and refer the reader to the illustrative example in Figure 4.2.

In that Figure the red-dashed arrows indicate the new links computed by one pointer-jumping step, and the table on the right of each list specifies the values of array $\texttt{Rank}[1, n]$ as they are recomputed after this step. The values in bold are the final/correct values. We notice that distances do not grow linearly (i.e. $1, 2, 3, \ldots$) but they grow as a power of two (i.e. $1, 2, 4, \ldots$), up to the step in which the next jump leads to reach t, the tail of the list. This means that the total number of times the parallel algorithm executes the two steps above is $O(\log n)$, thus resulting an exponential improvement with respect to the time required by the sequential algorithm. Given that n processors are involved, pointer-jumping executes a total of $O(n \log n)$ operations, which is inefficient if we compare it to the number $O(n)$ operations executed by the optimal RAM algorithm.

LEMMA 4.1 The parallel algorithm, using n processors and the pointer-jumping technique, takes $O(\log n)$ time and $O(n \log n)$ operations to solve the list-ranking problem.

Optimizations are possible to further improve the previous result and come close the optimal number of operations; for example, by *turning off* processors, as their corresponding items reach the end of the list, could be an idea but we will not dig into these details (see e.g. [2]) because they pertain to a course on parallel algorithms. Here we are interested in *simulating* the pointer-jumping technique in our setting which consists of one single processor and a 2-level memory, and show that deriving an I/O-efficient algorithm is very simple whenever an efficient parallel algorithm is

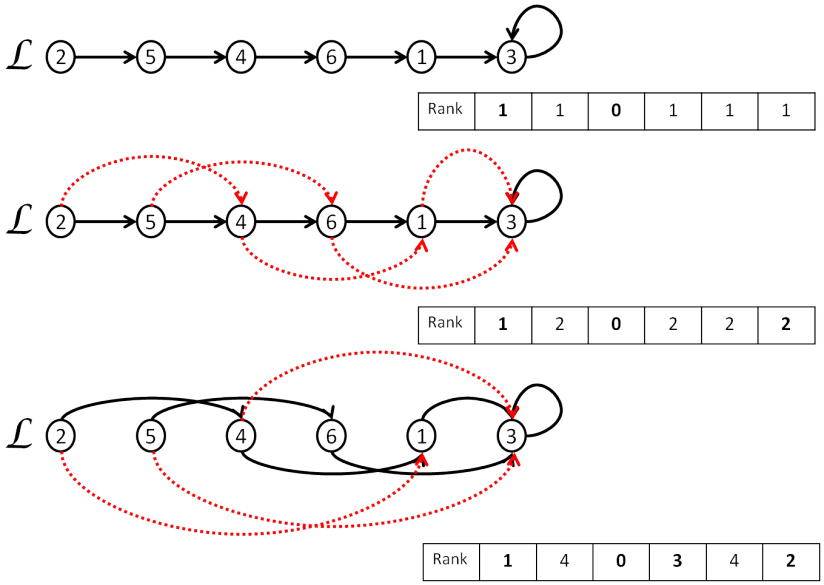


FIGURE 4.2: An example of pointer jumping applied to the list $\mathcal{L}$ of Figure 4.1. The dotted arrows indicate one pointer-jumping step applied onto the solid arrows, which represent the current configuration of the list.

available. The simplicity hinges onto an algorithmic scheme which deploys two basic primitives—Scan and Sort a set of triples— nowadays available in almost every distributed platforms, such as Apache Hadoop.

4.2 Parallel algorithm simulation in a 2-level memory

The key difficulty in using the pointer-jumping technique within the 2-level memory framework is the arbitrary layout of the list on disk, and the consequent arbitrary pattern of memory accesses to update `Succ`-pointers and `Rank`-values, which might induce many I/Os. To circumvent this problem we will describe how the two key steps of the pointer-jumping approach can be simulated via a constant number of `Sort` and `Scan` primitives over n triples of integers. Sorting is a basic primitive which is very much complicated to be implemented I/O-efficiently, and indeed will be the subject of the entire Chapter 5. For the sake of presentation, we will indicate its I/O-complexity as $\widetilde{O}(n/B)$ which means that we have hidden a logarithmic factor depending on the main parameters of the model, namely M, n, B. This factor is negligible in practice, since we can safely upper bound it with 4 or less, and so we prefer now to *hide it* in order to avoid jeopardizing the reading of this chapter. On the other hand, `Scan` is easy and takes $O(n/B)$ I/Os to process a contiguous disk portion occupied by the n triples.

We can identify a common algorithmic structure in the two steps of the pointer-jumping technique: each of them consists of an operation (either copy or sum) between two entries of an array (either `Succ` or `Rank`). For the sake of presentation we will refer to a generic array A, and model the parallel operation to be simulated on disk as follows:

Assume that a parallel step has the following form: $A[a_i]$ `op` $A[b_i]$*, where* `op` *is the operation executed in parallel over the two array entries* $A[a_i]$ *and* $A[b_i]$ *by all processors* $i = 1, 2, \ldots, n$ *which actually read* $A[b_i]$ *and use this value to update the content of* $A[a_i]$.

The operation `op` is a sum and an assignment for the update of the `Rank`-array (here A = `Rank`), it is a copy for the update of the `Succ`-array (here A = `Succ`). As far as the array indices are concerned they are, for both steps, $a_i = i$ and b_i = `Succ`$[i]$. The key issue is to show that $A[a_i]$ `op` $A[b_i]$ can be implemented, simultaneously over all $i = 1, 2, 3, \ldots, n$, by using a constant number of `Sort` and `Scan` primitives, thus taking a total of $\widetilde{O}(n/B)$ I/Os. The simulation consists of 5 steps:

1. `Scan` the disk and create a sequence of triples having the form $\langle a_i, b_i, 0\rangle$. Every triple brings information about the source address of the array-entry involved in `op` (b_i), its destination address (a_i), and the value that we are moving (the third component, initialized to 0).
2. `Sort` the triples according to their second component (i.e. b_i). This way, we are "aligning" the triple $\langle a_i, b_i, 0\rangle$ with the memory cell $A[b_i]$.
3. `Scan` the triples and the array A to create the new triples $\langle a_i, b_i, A[b_i]\rangle$. Notice that not all memory cells of A are referred as second component of any triple, nevertheless their coordinated order allows to copy $A[b_i]$ into the triple for b_i via a coordinated scan.
4. `Sort` the triples according to their first component (i.e. a_i). This way, we are aligning the triple $\langle a_i, b_i, A[b_i]\rangle$ with the memory cell $A[a_i]$.
5. `Scan` the triples and the array A and, for every triple $\langle a_i, b_i, A[b_i]\rangle$, update the content of the memory cell $A[a_i]$ according to the semantics of `op` and the value $A[b_i]$.

I/O-complexity is easy to derive since the previous algorithm is executing 2 `Sort` and 3 `Scan` involving n triples. Therefore we can state the following:

THEOREM 4.1 *The parallel execution of n operations* $A[a_i]$ `op` $A[b_i]$ *can be simulated in a* 2*-level memory model by using a constant number of* `Sort` *and* `Scan` *primitives, thus taking a total of* $\widetilde{O}(n/B)$ *I/Os.*

In the case of the parallel pointer-jumping algorithm, this parallel assignment is executed for $O(\log n)$ steps, so we have:

THEOREM 4.2 *The parallel pointer-jumping algorithm can be simulated in a* 2*-level memory model taking* $\widetilde{O}((n/B)\log n)$ *I/Os.*

This bound turns to be $o(n)$, and thus better than the direct execution of the sequential algorithm on disk, whenever $B = \omega(\log n)$. This condition is trivially satisfied in practice because $B \approx 10^4$ bytes and $\log n \leq 80$ for any real dataset size (being 2^{80} the number of atoms in the Universe[1]).

Figure 4.3 reports a running example of this simulation over the list at the top of the Figure 4.3. Table on the left indicates the content of the arrays `Rank` and `Succ` encoding the list; table on the right indicates the content of these two arrays after one step of pointer-jumping. The five columns of triples correspond to the application of the five `Scan`/`Sort` phases. This simulation is related to the update of the array `Rank`, array `Succ` can be recomputed similarly. Actually, the update

[1] See e.g. `http://en.wikipedia.org/wiki/Large_numbers`

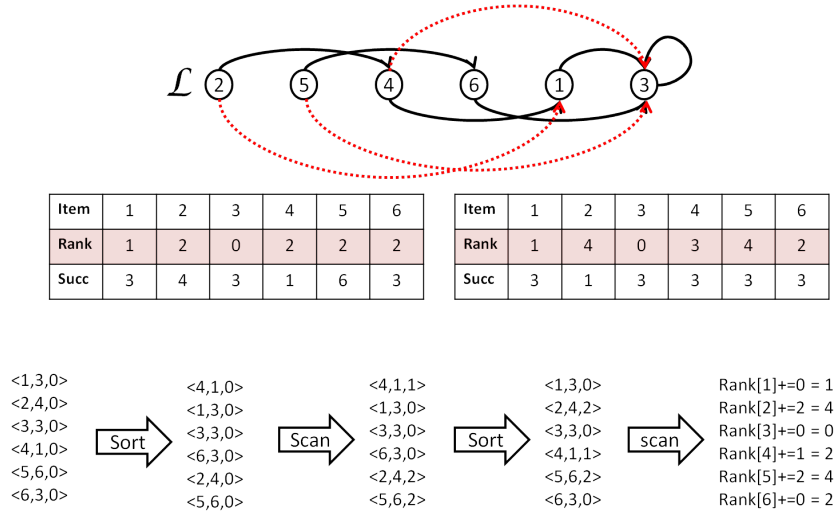


FIGURE 4.3: An example of simulation of the basic parallel step via `Scan` and `Sort` primitives, relative to the computation of the array `Rank`, with the configuration specified in the picture and tables above.

can be done simultaneously by using a quadruple instead of a triple which brings both the values of $\texttt{Rank}[\texttt{Succ}[i]]$ and the value of $\texttt{Succ}[\texttt{Succ}[i]]$, thus deploying the fact that both values use the same source and destination address (namely, i and $\texttt{Succ}[i]$).

The first column of triples is created as $\langle i, \texttt{Succ}[i], 0\rangle$, since $a_i = i$ and $b_i = \texttt{Succ}[i]$. The third column of triples is sorted by the second component, namely $\texttt{Succ}[i]$, and so its third component is obtained by `Scan`ning the array `Rank` and creating $\langle i, \texttt{Succ}[i], \texttt{Rank}[\texttt{Succ}[i]]\rangle$. The fourth column of triples is ordered by their first component, namely i, so that the final `Scan`-step can read in parallel the array `Rank` and the third component of those triples, and thus compute correctly $\texttt{Rank}[i]$ as $\texttt{Rank}[i] + \texttt{Rank}[\texttt{Succ}[i]] = 1 + \texttt{Rank}[\texttt{Succ}[i]]$.

The simulation scheme introduced in this section can be actually generalized to every parallel algorithm thus leading to the following important, and useful, result (see [1]):

THEOREM 4.3 *Every parallel algorithm using n processors and taking T steps can be simulated in a 2-level memory by a disk-aware sequential algorithm taking $\widetilde{O}((n/B)\ T)$ I/Os and $O(n)$ space.*

This simulation is advantageous whenever $T = o(B)$, which implies a sub-linear number of I/Os $o(n)$. This occurs in all cases in which the parallel algorithm takes a low *poly-logarithmic* time-complexity. This is exactly the situation of parallel algorithms developed over the so called P-RAM model of computation which assumes that all processors work independently of each other and they can access in constant time an unbounded shared memory. This is an ideal model which was very famous in the '80s-'90s and led to the design of many powerful parallel techniques, which have been then applied to distributed as well as disk-aware algorithms. Its main limit was to do not account for conflicts among the many processors accessing the shared memory, and a simplified communication among them. Nevertheless this simplified model allowed researchers to concentrate onto the algorithmic aspects of parallel computation and thus design precious parallel schemes as

the ones described below.

4.3 A Divide&Conquer approach

The goal of this section is to show that the list-ranking problem can be solved more efficiently than pointer-jumping on a list. The algorithmic solution we describe in this section relies on an interesting application of the *Divide&Conquer paradigm*, here specialized to work on a (mono-directional) list of items.

Before going into the technicalities related to this application, let us briefly recall the main ideas underlying the design of an algorithm, say $\mathcal{A}_{dc}$, based on the Divide&Conquer technique which solves a problem $\mathcal{P}$, formulated on n input data. $\mathcal{A}_{dc}$ consists of three main phases:

Divide. $\mathcal{A}_{dc}$ creates a set of k *subproblems*, say $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k$, having sizes $n_1, n_2, \ldots, n_k$, respectively. They are identical to the original problem $\mathcal{P}$ but are formulated on smaller inputs, namely $n_i < n$.

Conquer. $\mathcal{A}_{dc}$ is invoked *recursively* on the subproblems $\mathcal{P}_i$, thus getting the solution s_i.

Recombine. $\mathcal{A}_{dc}$ recombines the solutions s_i to obtain the solution s for the original problem $\mathcal{P}$. s is *returned* as output of the algorithm.

It is clear that the Divide&Conquer technique originates a *recursive* algorithm $\mathcal{A}_{dc}$, which needs a *base case* to terminate. Typically, the base case consists of stopping $\mathcal{A}_{dc}$ whenever the input consists of few items, e.g. $n \leq 1$. In these small-input cases the solution can be computed easily and directly, possibly by enumeration.

The time complexity $T(n)$ of $\mathcal{A}_{dc}$ can be described as a recurrence relation, in which the base condition is $T(n) = O(1)$ for $n \leq 1$, and for the other cases it is:

$$T(n) = D(n) + R(n) + \sum_{i=1,\ldots,k} T(n_i)$$

where $D(n)$ is the cost of the Divide step, $R(n)$ is the cost of the Recombination step, and the last term accounts for the cost of all recursive calls. These observations are enough for these notes; we refer the reader to Chapter 4 in [3] for a deeper and clean discussion about the Divide&Conquer technique and the Master Theorem that provides a mathematical solution to recurrence relations, such as the one above.

We are ready now to specialize the Divide&Conquer technique over the List-Ranking problem. The algorithm we propose is pretty simple and starts by assigning to each item i the value $\texttt{Rank}[i] = 0$ for $i = t$, otherwise $\texttt{Rank}[i] = 1$. Then it executes three main steps:

Divide. We identify a set of items $I = \{i_1, i_2, \ldots, i_h\}$ drawn from the input list $\mathcal{L}$. Set I must be an *independent set*, which means that the successor of each item in I does not belong to I. This condition clearly guarantees that $|I| \leq n/2$, because at most one item out of two consecutive items may be selected. The algorithm will guarantee also that $|I| \geq n/c$, where $c > 2$, in order to make the approach effective.

Conquer. Form the list $\mathcal{L}^* = \mathcal{L} - I$, by pointer-jumping only on the predecessors of the removed items I: namely, for every $\texttt{Succ}[x] \in I$ we set $\texttt{Rank}[x] += \texttt{Rank}[\texttt{Succ}[x]]$ and $\texttt{Succ}[x] = \texttt{Succ}[\texttt{Succ}[x]]$. This way, at any recursive call, $\texttt{Rank}[x]$ accounts for the number of items of the original input list that lie between x and the current $\texttt{Succ}[x]$. Then solve recursively the list-ranking problem over $\mathcal{L}^*$. Notice that $n/2 \leq |\mathcal{L}^*| \leq (1 - 1/c)n$, so that the recursion acts on a list which is a fractional part of $\mathcal{L}$. This is crucial for the efficiency of the recursive calls.

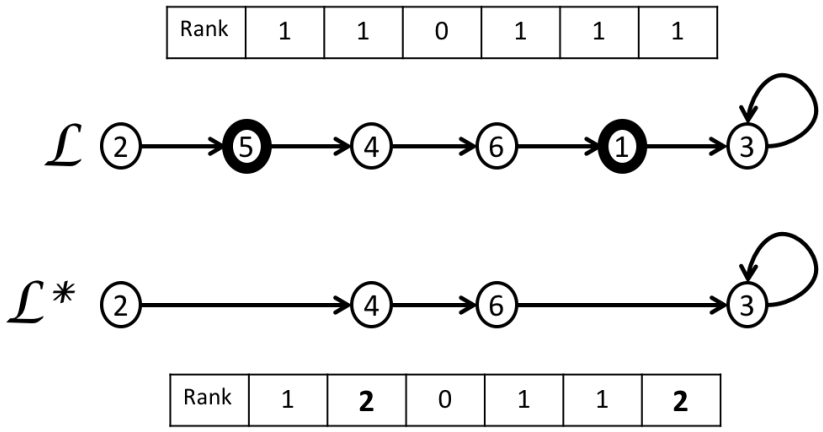


FIGURE 4.4: An example of reduction of a list due to the removal of the items in an Independent Set, here specified by the bold nodes. The new list on the bottom is the one resulting from the removal, the Rank-array is recomputed accordingly to reflect the missing items. Notice that the Rank-values are 1, 0 for the tail, because we assume that $\mathcal{L}$ is the initial list.

Recombine. At this point we can assume that the recursive call has computed correctly the list-ranking of all items in $\mathcal{L}^*$. So, in this phase, we derive the rank of each item $x \in I$ as $\texttt{Rank}[x] = \texttt{Rank}[x] + \texttt{Rank}[\texttt{Succ}[x]]$, by adopting an update rule which reminds the one used in pointer jumping. The correctness of Rank-computation is given by two facts: (i) the independent-set property about I ensures that $\texttt{Succ}[x] \notin I$, thus $\texttt{Succ}[x] \in \mathcal{L}^*$ and so its Rank is available; (ii) by induction, $\texttt{Rank}[\texttt{Succ}[x]]$ accounts for the distance of $\texttt{Succ}[x]$ from the tail of $\mathcal{L}$ and $\texttt{Rank}[x]$ accounts for the number of items between x and $\texttt{Succ}[x]$ in the original input list (as observed in Conquer's step). In fact, the removal of x (because of its selection in I, may occur at any recursive step so that x may be far from the current $\texttt{Succ}[x]$ when considered the original list; this means that it might be the case of $\texttt{Rank}[x] \gg 1$, which the previous summation-step will take into account. As an example, Figure 4.4 depicts the starting situation in which all ranks are 1 except the tail's one, so the update is just $\texttt{Rank}[x] = 1 + \texttt{Rank}[\texttt{Succ}[x]]$. In a general recursive step, $\texttt{Rank}[x] \geq 1$ and so we have to take care of this when updating its value. As a result all items in $\mathcal{L}$ have their Rank-value correctly computed and, thus, induction is preserved and the algorithm may return to its invoking caller.

Figure 4.4 illustrates how an independent set (denoted by bold nodes) is removed from the list $\mathcal{L}$ and how the Succ-links are properly updated. Notice that we are indeed pointer-jumping only on the predecessors of the removed items (namely, the predecessors of the items in I), and that the other items leave untouched their Succ-pointers. It is clear that, if the next recursive step selects $I = \{6\}$, the final list will be constituted by three items $\mathcal{L} = (2, 4, 3)$ whose final ranks are $(6, 3, 0)$, respectively. The Recombination-step will re-insert 6 in $\mathcal{L} = (2, 4, 3)$, just after 4, and compute $\texttt{Rank}[6] = \texttt{Rank}[6] + \texttt{Rank}[3] = 2 + 0 = 2$ because $\texttt{Succ}[6] = 3$ in the current list. Conversely, if one would have not taken into account the fact that item 6 may be far from its current $\texttt{Succ}[6] = 3$, when referred to the original list, and summed 1 it would have made a wrong calculation for $\texttt{Rank}[6]$.

This algorithm makes clear that its I/O-efficiency depends onto the Divide-step. In fact, Conquer-step is recursive and thus can be estimated as $T((1 - \frac{1}{c})n)$ I/Os; Recombine-step executes all re-

insertions simultaneously, given that the removed items are not contiguous (by definition of independent set), and can be implemented by Theorem 4.1 in $\widetilde{O}(n/B)$ I/Os.

THEOREM 4.4 *The list-ranking problem formulated over a list $\mathcal{L}$ of length n, can be solved via a Divide&Conquer approach taking $T(n) = I(n) + \widetilde{O}(n/B) + T((1 - \frac{1}{c})n)$ I/Os, where $I(n)$ is the I/O-cost of selecting an independent set from $\mathcal{L}$ of size at least n/c (and, of course, at most $n/2$).*

Deriving a large independent set is trivial if a scan of the list $\mathcal{L}$ is allowed, just pick one every two items. But in our disk-context the list scanning is I/O-inefficient and this is exactly what we want to avoid: otherwise we would have solved the list-ranking problem!

In what follows we will therefore concentrate on the problem of identifying a *large* independent set within the list $\mathcal{L}$. The solution must deploy only local information within the list, in order to avoid the execution of many I/Os. We will propose two solutions: one is simple and randomized, the other one is deterministic and more involved. It is surprising that the latter technique (called *deterministic coin tossing*) has found applications in many other contexts, such as data compression, text similarity, string-equality testing. It is a very general and powerful technique that, definitely, deserves some attention in these notes.

4.3.1 A randomized solution

The algorithmic idea, as anticipated above, is simple: toss a fair coin for each item in $\mathcal{L}$, and then select those items i such that $\texttt{coin}(i) = \texttt{H}$ but $\texttt{coin}(\texttt{Succ}[i]) = \texttt{T}$.[2]

The probability that the item i is selected is $\frac{1}{4}$, because this happens for one configuration (HT) out of the four possible configurations. So the average number of items selected for I is $n/4$. By using sophisticated probabilistic tools, such as Chernoff bounds, it is possible to prove that the number of selected items is strongly concentrated around $n/4$. This means that the algorithm can repeat the coin tossing until $|I| \geq n/c$, for some $c > 4$. The strong concentration guarantees that this repetition is executed a (small) constant number of times.

We finally notice that the check on the values of `coin`, for selecting I's items, can be simulated by Theorem 4.1 via few `Sort` and `Scan` primitives, thus taking $I(n) = \widetilde{O}(n/B)$ I/Os on average. So, by substituting this value in Theorem 4.4, we get the following recurrence relation for the I/O-complexity of the proposed algorithm: $T(n) = \widetilde{O}(n/B) + T(\frac{3n}{4})$. It can be shown by means of the Master Theorem (see Chapter 4 in [3]) that this recurrence relation has solution $\widetilde{O}(n/B)$.

THEOREM 4.5 *The list-ranking problem, formulated over a list $\mathcal{L}$ of length n, can be solved with a randomized algorithm in $\widetilde{O}(n/B)$ I/Os on average.*

4.3.2 Deterministic coin-tossing$^{\infty}$

The key property of the randomized process was the *locality* of I's construction which allowed to pick an item i by just looking at the results of the coins tossed for i itself and for its successor $\texttt{Succ}[i]$. In this section we try to simulate *deterministically* this process by introducing the so called *deterministic coin-tossing* strategy that, instead of assigning two coin values to each item (i.e. `H` and `T`), it starts by assigning n coin values (hereafter indicated with the integers $0, 1, \ldots, n-1$) and

[2] The algorithm works also in the case that we exchange the role of head (H) and tail (T); but it does not work if we choose the configurations HH or TT. Why?

eventually reduces them to *three* coin values (namely $0, 1, 2$). The final selection process for I will then pick the items whose coin value is minimum among their adjacent items in $\mathcal{L}$. Therefore, here, three possible values and three possible items to be compared, still a constant execution of `Sort` and `Scan` primitives.

The pseudo-code of the algorithm follows.

Initialization. Assign to each item i the value $\texttt{coin}(i) = i - 1$. This way all items take a different coin value, which is smaller than n. We represent these values in $b = \lceil \log n \rceil$ bits, and we denote by $\texttt{bit}_b(i)$ the binary representation of $\texttt{coin}(i)$ using b bits.

Get 6-coin values. Repeat the following steps until $\texttt{coin}(i) < 6$, for all i:

- Compute the position $\pi(i)$ where $\texttt{bit}_b(i)$ and $\texttt{bit}_b(\texttt{Succ}[i])$ differ, and denote by $z(i)$ the bit-value of $\texttt{bit}_b(i)$ at that position.
- Compute the new coin-value for i as $\texttt{coin}(i) = 2\pi(i) + z(i)$ and set the new binary-length representation as $b = \lceil \log b \rceil + 1$.

Get just 3-coin values. For each element i, such that $\texttt{coin}(i) \in \{3, 4, 5\}$, do $\texttt{coin}(i) = \{0, 1, 2\} - \{\texttt{coin}(\texttt{Succ}[i]), \texttt{coin}(\texttt{Pred}[i])\}$.

Select I. Pick those items i such that $\texttt{coin}(i)$ is a local minimum, namely it is smaller than $\texttt{coin}(\texttt{Pred}[i])$ and $\texttt{coin}(\texttt{Succ}[i])$.

Let us first discuss the correctness of the algorithm. At the beginning all coin values are distinct, and in the range $\{0, 1, \ldots, n-1\}$. By distinctness, the computation of $\pi(i)$ is sound and $2\pi(i) + z(i) \leq 2(b-1)+1 = 2b-1$ since $\texttt{coin}(i)$ was represented with b bits and hence $\pi(i) \leq b-1$ (counting from 0). Therefore, the new value $\texttt{coin}(i)$ can be represented with $\lceil \log b \rceil + 1$ bits, and thus the update of b is correct too.

A key observation is that the new value of $\texttt{coin}(i)$ is still different of the coin value of its adjacent items in $\mathcal{L}$, namely $\texttt{coin}(\texttt{Succ}[i])$ and $\texttt{coin}(\texttt{Pred}[i])$. We prove it by contradiction. Let us assume that $\texttt{coin}(i) = \texttt{coin}(\texttt{Succ}[i])$ (the other case is similar), then $2\pi(i)+z(i) = 2\pi(\texttt{Succ}[i])+z(\texttt{Succ}[i])$. Since z denotes a bit value, the two coin-values are equal iff it is both $\pi(i) = \pi(\texttt{Succ}[i])$ and $z(i) = z(\texttt{Succ}[i])$. But if this condition holds then the two bit sequences $\texttt{bit}_b(i)$ and $\texttt{bit}_b(\texttt{Succ}[i])$ cannot differ at bit-position $\pi(i)$.

Easily it follows the correctness of the step which allows to go from 6-coin values to 3-coin values, as well as it is immediate the proof that the selected items form an independent set because of the minimality of $\texttt{coin}(i)$ and distinctness of adjacent coin values.

As far as the I/O-complexity is concerned, we start by introducing the function $\log^* n$ defined as $\min\{j \mid \log^{(j)} n \leq 1\}$, where $\log^{(j)} n$ is the repeated application of the logarithm function for j times to n. As an example[3] take $n = 16$ and compute $\log^{(0)} 16 = 16, \log^{(1)} 16 = 4, \log^{(2)} 16 = 2, \log^{(3)} 16 = 1$; thus $\log^* 16 = 3$. It is not difficult to convince yourselves that $\log^* n$ grows very much slowly, and indeed its value is 5 for $n = 2^{65536}$.

In order to estimate the I/O-complexity, we need to bound the number of iterations needed by the algorithm to reduce the coin-values to $\{0, 1, \ldots, 5\}$. This number is $\log^* n$, because at each step the reduction in the number of possible coin-values is logarithmic ($b = \lceil \log b \rceil + 1$). All single steps can be implemented by Theorem 4.1 via few `Sort` and `Scan` primitives, thus taking $\widetilde{O}(n/B)$ I/Os. So the construction of the independent set takes $I(n) = \widetilde{O}((n/B) \log^* n) = \widetilde{O}(n/B)$ I/Os, by definition of $\widetilde{O}()$. The size of I can be lower bounded as $|I| \geq n/4$ because the distance between two consecutive

[3] Recall that logarithms are in base 2 in these lectures.

selected items (local minima) is maximum when the coin-values form a *bitonic sequence* of the form $\ldots, 0, 1, 2, 1, 0, \ldots$.

By substituting this value in Theorem 4.4 we get the same recurrence relation of the randomized algorithm, with the exception that now the I/O-bound is worst case and deterministic: $T(n) = \widetilde{O}(n/B) + T(\frac{3n}{4})$.

THEOREM 4.6 *The list-ranking problem, formulated over a list $\mathcal{L}$ of length n, can be solved with a deterministic algorithm in $\widetilde{O}(n/B)$ worst-case I/Os.*

A comment is in order to conclude this chapter. The logarithmic term hidden in the $\widetilde{O}()$-notation has the form $(\log^* n)(\log_{M/B} n)$, which can be safely assumed to be smaller than 15 because, in practice, $\log_{M/B} n \leq 3$ and $\log^* n \leq 5$ for n up to 1 petabyte.

References

[1] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, Jeffrey Scott Vitter. External-Memory Graph Algorithms. *ACM-SIAM Symposium on Algorithms (SODA)*, 139-149, 1995.

[2] Joseph JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[3] Tomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[4] Jeffrey Scott Vitter. Faster methods for random sampling. *ACM Computing Surveys*, 27(7):703–718, 1984.

5

Sorting Atomic Items

This lecture will focus on the very-well known problem of sorting a set of *atomic* items, the case of *variable-length* items (aka strings) will be addressed in the following chapter. Atomic means that they occupy a constant-fixed number of memory cells, typically they are integers or reals represented with a fixed number of bytes, say 4 (32 bits) or 8 (64 bits) bytes each.

The sorting problem. *Given a sequence of n atomic items $S[1,n]$ and a total ordering $\leq$ between each pair of them, sort S in increasing order.*

We will consider two complemental sorting paradigms: the *merge-based* paradigm, which underlies the design of Mergesort, and the *distribution-based* paradigm which underlies the design of Quicksort. We will adapt them to work in the 2-level memory model, analyze their I/O-complexities and propose some useful tools that can allow to speed up their execution in practice, such as the Snow Plow technique and Data compression. We will also demonstrate that these disk-based adaptations are *I/O-optimal* by proving a sophisticated lower-bound on the number of I/Os any external-memory sorter must execute to produce an ordered sequence. In this context we will relate the Sorting problem with the so called *Permuting* problem, typically neglected when dealing with sorting in the RAM model.

The permuting problem. *Given a sequence of n atomic items $S[1,n]$ and a permutation $\pi[1,n]$ of the integers $\{1,2,\ldots,n\}$, permute S according to π thus obtaining the new sequence $S[\pi[1]], S[\pi[2]], \ldots, S[\pi[n]]$.*

Clearly Sorting includes Permuting as a sub-task: to order the sequence S we need to determine its sorted permutation and then implement it (possibly these two phases are intricately intermingled). So Sorting should be more difficult than Permuting. And indeed in the RAM model we know that sorting n atomic items takes $\Theta(n \log n)$ time (via Mergesort or Heapsort) whereas permuting them takes $\Theta(n)$ time. The latter time bound can be obtained by just moving one item at a time according to what indicates the array π. Surprisingly we will show that this *complexity gap* does