Praktikum Sekundärspeicheralgorithmen
Algorithmik II

Sanders, Dementiev, Schultes, Singler   SS 06

http://algo2.iti.uka.de/iopraktO6.php

Experiment 4

Deadline: 13:00 — June 19, 2006

1 List Ranking

Let L be a linked list, i.e., a collection of nodes $x_1, \ldots, x_n$ such that each node x_i, except the tail of the list, stores a pointer $\mathrm{succ}(x_i)$ to its successor in L, no two nodes have the same successor, and every node can reach the tail of L by following successor pointers. The list L can be also viewed as a directed graph with nodes $V = \{x_1, \ldots, x_n\}$ and edges $E = \{(x_i, succ(x_i)) : x_i \textit{ is not the tail}\}$. The list ranking problem is that of computing for every node x_i, its distance from the tail (or head) of L, i.e., the number of edges on the path from the head (tail) of L to x_i.

1.1 Algorithm of Sibeyn

The algorithm of Sibeyn [Sib] for I/O-efficient list ranking is depicted in Figure 1. The algorithm accepts a list (or an array) of edges L: directed edge $(u, v) \in L$ iff node v is the successor of node u. In the first step, the edges in the list are scanned and added to an adjacency list together with the distances between incident nodes (positive: outgoing, negative: incoming). For edge (u, v) with $u < v$, node u is added to the list of v with distance -1, otherwise v is added to the list of u with distance $+1$. After that the algorithm makes two passes through the nodes: one starting with node $n - 1$ and ending with node 1, and another one running in the reverse direction.

```
**foreach** edge $(u,v) \in L$                    // Step 1
    **if** $u < v$ **then** add $(u,-1)$ to the list of $v$
    **else** add $(v,1)$ to the list of $u$
**for** $u := n-1$ **downto** 1 **do**             // Step 2
    **if** $u$ has list entries $(v,l_v)$ and $(w,l_w)$, $v < w < u$ **then**
        add $(v, l_v - l_w)$ to the list of $w$
    **else** // $u$ has a single entry $(w,l_w)$, $w < u$
        add $(-\infty, -l_w)$ to the list of $w$
    $ref_u := w$
    $\delta_u := l_w$
// node 0 has list entries $(-\infty, l_{head})$, $l_{head} < 0$ and/or $(-\infty, l_{tail})$, $l_{tail} > 0$
$d(0) := l_{last}$ or 0 if node 0 is the tail        // distance from the last node
**for** $u := 1$ **to** $n-1$ **do**                // Step 3
    $d(u) := d(ref_u) + \delta_u$
```

Figure 1: Sibeyn's list ranking algorithm.

During a right-to-left pass (step 2), for each node u its list is inspected: if it has two entries (v, l_v) and (w, l_w), $v < w < u$, then the length of the path between nodes v and w is computed $(l_v - l_w)$ and entry $(v, l_v - l_w)$ is added to the list of w. Otherwise, node u is the head or tail of L and has only a single entry (w, l_w), $w < u$. In this case, we add an entry $(-\infty, l_w)$ to the list of w. These operations are equivalent to removing node u and inserting the direct link between its neighbor nodes. An example is illustrated in Figure 2. The table on the left shows the lists of the nodes, each row represents one iteration. After Step 2, the list of node 0 contains entries $(-\infty, l_{head})$, $l_{head} < 0$ and/or $(-\infty, l_{tail})$, $l_{tail} > 0$. The list might contain only one of these entries if node 0 is the tail or the head of L. In any case, the distance of node 0 from the tail of the list can be computed. Step 3 processes the nodes in reverse order and computes the distance of node u from the distance of already processed nodes $d(ref_u)$. The reference node ref_u and the length of the path δ_u from u to ref_u are remembered in Step 2.

1.2 Implementation details

- To store the lists of nodes, use a single I/O-efficient STXXL priority queue. Entry (v, l_v) in the list of node u can be represented as a record (u, v, l_v) in a priority queue ordered by field u.
- I/O-efficient STXXL stack is a proper data structure to keep δ_u values.
- Remembering ref_u values can be done by associating with each node w a list of nodes, which use w as a reference, we call them send lists. Then the assignment $ref_u = w$ can be translated to adding u to the send list of w. The send lists can be implemented I/O-efficiently using a single STXXL priority queue in the same way as the adjacency lists.
- Efficient delivery of $d(ref_u)$ values can be achieved with another STXXL priority queue that keeps values $(u, d(ref_u))$ ordered by u (distance messages). In each iteration of Step 3 this value is extracted from the priority queue (the min element) to be used in the computation of $d(u)$. Then for all v in the send list of u the entries $(v, d(u))$ have to be inserted in the distance message priority queue.

Exercise 1
Prove that the I/O-efficient implementation of the Sibeyn's link ranking algorithm runs in $\mathcal{O}(\text{sort}(|L|))$ I/Os.

2 Random List Generation

A random list L with n nodes can be generated I/O-efficiently using an I/O-efficient random permutation:

1. Compute $R = \{r_0, r_1, \ldots, r_{n-1}\}$, a random permutation of the sequence $\{0, 1, \ldots, n-1\}$,
2. Compute sequence $S = \{s_0, s_1, \ldots, s_{n-2}\}$, with pairs $s_i = (r_i, r_{i+1})$, $0 \le i \le n-2$.
3. The edge list L will be the sequence S sorted by the first component of the pairs (edges).

Exercise 2
Implement the Sibeyn's list ranking algorithm using STXXL.

The prototype of the function must be the following:

```
template <class InIterator_, class OutIterator_>
stxxl::int64 list_rankingV1(
        InIterator_ first, InIterator_ beyond, // input range
        OutIterator_ result,// output iterator
        unsigned M); // in bytes
```

The input range `[first,beyond)` is the range in a container with the edges of input list L. `out` is the output iterator for storing the output values $(u, d(u))$, where u is the id of a node, and $d(u)$ is the distance of the node to the tail of the list. The distance of the tail node is zero. `M` is the number of internal memory bytes the function can use. This memory is to be shared between the STXXL data structures you will use (stacks, priority queues, vectors). The function returns the (id of the) tail node.

Result checking: Check the correctness of the output of your I/O-efficient implementation with the help of the unsorted sequence S used in the input generation.

1	2	3	4	5	6
	(1,1)	(2,1)		(1,-1)	(5,-1) (4,1)
	(1,1)	(2,1)		(1,-1) (4,2)	
	(1,1)	(2,1)	(1,-3)		
$(-\infty,3)$	(1,1)	(2,1)			
$(-\infty,3)$	(1,1) $(-\infty,-1)$				
$(-\infty,3)$ $(-\infty,-2)$					

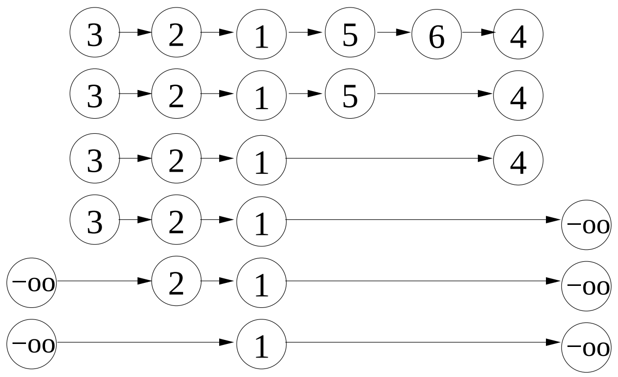

Figure 2: Running Step 2 of Sibeyn's algorithm on list $(3 \rightarrow 2 \rightarrow 1 \rightarrow 5 \rightarrow 6 \rightarrow 4)$.

Exercise 3
Measurements:

- Choose $M = 1\,\mathrm{GB}$.
- Use `int` as the node type.
- Measure the running time of your list ranking function for random lists with n nodes, where $n = 2^{20}, 2^{21}, \ldots, 2^{30}$ (powers of 2). Also, measure the I/O wait time [1] using `stxxl::stats` class. Try to tune your implementation changing the block sizes, the distribution of given memory M between the data structures and block pools. Find the best values.
- For generating inputs for your implementation, use the algorithm above with the `random_shuffleV2` function from Experiment 2.
- Download the bucket implementation of a different list ranking algorithm [Sib97]: Vitaly Osipov's implementation `http://algo2.iti.uka.de/dementiev/courses/ioprakt06/lr.tgz`
- Run the bucket implementation for the same input range $n = 2^{20}, 2^{21}, \ldots, 2^{30}$ (change the first command line argument). It is an STXXL implementation as well, so make sure it uses the same STXXL external files as your implementation. Measure the running time and the I/O wait time of *list ranking*.

Write a report that includes *at least* the following:

- The analysis for Exercise 1.
- The running time and I/O wait time of your list ranking implementation drawn in a single figure (plot, as usual, time per input item).
- The running time and I/O wait time of the bucket list ranking implementation drawn in a single figure.
- A plot with running times of your implementation and the bucket implementation together.
- Discussion of the results: which algorithm is faster, possible reasons (I/O wait time values might help to explain).

Send your source code and your report with figures to `dementiev@ira.uka.de` before the deadline. Also, make an appointment with Roman Dementiev for the defense of your work.

References

[Sib] Jop F. Sibeyn. External connected components. In *SWAT 2004*, pages 468–479.

[Sib97] J.F. Sibeyn. From parallel to external list ranking. Technical Report MPI-I-97-1-021, Max-Planck Institut für Informatik, Saarbrücken, Germany, 1997.

[1] I/O wait time is the number of seconds the processing thread has been waiting for I/O completion. This value indicates whether the application is I/O or compute bound. A call of `stats::reset()` starts a measurement of I/O wait time.