Praktikum Sekundärspeicheralgorithmen
Algorithmik II

Sanders, Dementiev, Schultes, Singler   SS 06

http://algo2.iti.uka.de/ioprakt06.php

Experiment 3

Deadline: 13:00 — June 6, 2006

Random permutations.

The goal of this experiment is to implement the functionality of `std::random_shuffle` [SGI] I/O-efficiently using Stxxl.

Exercise 1
Find the worst case I/O-complexity of the `std::random_shuffle` algorithm analyzing the source code [SGI].

Exercise 2
Design and implement an I/O-efficient algorithm that checks whether a given iterator range of an `stxxl::vector` with integers is a permutation of $\{0, \ldots, n-1\}$. The worst case I/O complexity must be $\mathcal{O}(\mathrm{sort}(n))$ I/Os. The (ordering of the) input can be destroyed.

Exercise 3
Implement an I/O-efficient algorithm that generates a random permutation of a sequence of n integers (`stxxl::int64`). The input sequence is a permutation of integer numbers in the range $\{0, \ldots, n-1\}$.

The implementation can assume that the arguments are iterators of an `stxxl::vector`. The prototype of the function should be:

```
template <typename ExtIterator_>
void random_shuffleV1(ExtIterator_ first, ExtIterator_ beyond, unsigned M)
```

M is the number of internal memory bytes the implementation is allowed to use.

To randomly permute the input, your algorithm assigns a random key $\in \{0, \ldots, n-1\}$ to each of the input elements and then sorts the elements using this key. To avoid the additional I/O-volume caused by keeping the random keys together with each element, we will compute the *stable* random keys on the fly when they are needed for a comparison in the sorting: the key of input integer element e equals $\delta(e)$, where $\delta : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ is a *Feistel* permutation. A C++ class that implements the Feistel permutations is available at `http://algo2.iti.uka.de/dementiev/courses/ioprakt06/feistel.tgz`.

Implement the proposed algorithm using `stxxl::sort` with a comparison functor that compares two elements based on their δ value. Make sure that the permutations of large ($> 2 \cdot M$) and small (in-memory) inputs obtained with this implementation are valid.

Exercise 4
Implement a generic I/O-efficient random permutation algorithm without any assumptions about the input element type.

The prototype of the function should be:

```
template <typename ExtIterator_>
void random_shuffleV2(ExtIterator_ first, ExtIterator_ beyond, unsigned M)
```

The algorithm to implement is a variant of [San98]. The input is scanned and the elements are distributed into $k = \mathcal{O}(M/B)$ random buckets. Then, each bucket is read into internal memory of size $\Theta(M)$. Each bucket is permuted internally and written to the output. If a bucket does not fit into internal memory then it is permuted recursively.

Implementation details:

- The buckets should be implemented as `stxxl::grow_shrink_stack2`(s) (see `stxxl::STACK_GENERATOR`).
- To achieve the best overlapping between I/O and computation, the common `stxxl::write_pool` and `stxxl::prefetch_pool` pools will be used. These two pools will be shared between all stacks [1].
- To make sufficient space for the overlap buffers, the block size B of the stacks and the number of buckets k should be chosen such that:
  1. B is a power of two: 2 MB, 1 MB, 512 KB.
  2. $k = \frac{M}{3B}$
- Before the distribution phase, the write pool is created with $\frac{2M}{3B}$ blocks and the prefetch pool is created with 0 blocks.
- After the distribution phase the write pool is resized to 0 blocks, the prefetch pool to 4 blocks. At this moment, $\frac{M}{3B}$ blocks are still occupied by the tails of the stacks. The remaining memory ($\frac{2M}{3} - 4B$ bytes) can be used for bucket permutation.
- Permute each bucket (before the bucket permutation, call the `set_prefetch_aggr` function of the stack with parameter value 4 to read ahead 4 bucket blocks):
  – If a bucket fits in $\frac{2M}{3} - 4B$ bytes, then create an array of required size, read the elements into it, permute internally using `std::random_shuffle`, and write the result to the input/output iterator range.
  – If a bucket does not fit into the $\frac{2M}{3} - 4B$ bytes (which will be very unlikely), copy the content of the stack to a temporary `stxxl::vector`, permute it recursively using $\frac{2M}{3} - 4B$ bytes. Copy the result to the input/output iterator range.
- For reading the input and writing the output use Stxxl streaming components for the best I/O efficiency. You can use the code available at `http://algo2.iti.uka.de/dementiev/courses/ioprakt06/shuffleV2.cpp`.

Make sure that the permutations of large ($> 2 \cdot M$) and small (in-memory) inputs obtained with this implementation are valid.

Exercise 5

Measurements:

- Permute the sequences $\{0, \ldots, n-1\}$ of type `stxxl::int64` stored in `stxxl::vector`.
- Choose $M = 1$ GB.
- For performance measurements compile your code without debug options with the highest optimization level settings, i. e. for `g++` use options `-DNDEBUG -O3`.
- Measure only the time for the permutation itself. Repeat measurements to achieve better accuracy.

[1] To create a stack that uses single shared pools use its special constructor with pool parameters.

- Run both I/O-efficient permuting algorithms for the following inputs: 128 MB, 256 MB, 512 MB, 1 GB, 2 GB, 4 GB, 8 GB, 16 GB, (32 GB — optional). Make sure you have enough space in the external STXXL file (check `.stxxl` configuration file and the output of the `df` command). For inputs 128 MB, 256 MB, 512 MB, 1 GB measure also the running time of `std::random_shuffle` working on `std::vector`. Draw plots with the graphs for the three values of B for the external memory algorithms and the graph for `std::random_shuffle`. It is recommended to draw time per input element on the y-axis.

Write a report that should include at least the following:

- The analysis for Exercise 1.
- The algorithm description and the analysis from Exercise 2.
- The plots with *detailed* explanations: Why one algorithm is faster than another for small/middle/large inputs.

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

[San98] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67(6):305–310, 1998.

[SGI] SGI. STL documentation: The `random_shuffle` algorithm. `http://www.sgi.com/tech/stl/random_shuffle.html` Source code: `http://www.irisa.fr/siames/OpenMASK/documentation/stl/stl__algo_8h-source.html#l00622`.