Sanders, Dementiev, Schultes, Singler SS 06

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

Experiment 2

Deadline: 13:00 — May 22, 2006

Exercise 1
A matrix class

Implement an external memory (dense) matrix class that uses a single `stxxl::vector` to store the content in row major order. For a $(m \times n)$-dimensional matrix and $0 \le i < m$, $0 \le j < n$ every element $m_{i,j}$ is mapped to the $(i \cdot n + j)$-th element of the vector container. The class must be configurable by two template parameters: `matrix<T,B>`, where `T` is the element data type and `B` is the vector block size in bytes. The `stxxl::vector` should have only one page having one block. The block size is a tuning parameter.

The class must implement the following members:

Member	Description
size_type	Type of size (must be `stxxl::int64`)
value_type	Type of element (must be `T`)
reference	Type of reference to an element (must be `T &`)
const_reference	Type of const reference to an element (must be `const T &`)
matrix (size_type size1, size_type size2)	Allocates an uninitialized matrix that holds size1 rows of size2 elements.
size_type size1 () const	Returns the number of rows.
size_type size2 () const	Returns the number of columns.
const_reference operator () (size_type i, size_type j) const	Returns a const reference of the j-th element in the i-th row.
reference operator () (size_type i, size_type j)	Returns a reference of the j-th element in the i-th row.

Exercise 2
I/O-efficient matrix transposition

Implement I/O-efficient matrix transposition using the matrix class from the previous exercise. The algorithm is described in [CS] (Section 3.2 – Algorithm 2). The prototype of the transpose function must be the following:

```
template <class T1, class T2, unsigned B1, unsigned B2>
void transpose(matrix<T1,B1> & C, const matrix<T2,B2> & A, unsigned M)
```

where `A` is the input matrix, `C` is the output matrix, `M` is the number of internal memory bytes that transpose function is allowed to use for holding the sub-matrix. The sub-matrix can be represented as a usual C++ array of arrays of type `T1` [1] with a rows and b columns, such that $a \cdot b \cdot$`sizeof(T1)`$\le M$. The matrix transposition of the sub-matrix must be done in-place.

Implement the internal memory matrix transposition algorithm (mentioned in the lecture) that transposes internal matrix `A` to internal matrix `B`. To represent the matrices use dynamically allocated C++ arrays of arrays of type `T`.

In the experiments you will compare the internal memory implementation with the I/O-efficient implementation. Use C++ type `double` as the matrix element type. The test programs should do the following (in both versions):

[1] Dynamically allocated using `new T*[a]` and `new T[b]`.

1. Create matrix A of size $(N \times 2N)$ and matrix B of size $(2N \times N)$.
2. Fill elements of **both** matrices with arbitrary values (e. g. $i - j$ where i is the row number and j is the column number).
3. Start the time measurement.
4. Perform the matrix transposition.
5. Stop the time measurement.
6. Output the total measured time t and the time per matrix element, i. e. $t/(2N^2)$.

The choice of parameters:

- The experiments should be done for at least $N = 1000, 2000, 4000, 6000, 7000, 8000, 9000, 16000$. For the internal algorithm perform the measurements only until the point when the system starts to thrash, for example, $N = 16000$ can be excluded. For I/O-efficient implementations make sure that your STXXL external memory is at least 8GB large to experiment with larger inputs.
- The memory size M assigned to the `transpose` function should be set to 1 GByte (=1073741824 bytes).
- The experiments for the I/O-efficient version should include measurements for the following `stxxl::vector` block sizes: 32 KBytes, 64KBytes, 128KBytes.
- The dimensions of the in-memory sub-matrix should be chosen such that $a, b \in \{(B/4)/\cdot sizeof(T1)), \ldots, (4B)/sizeof(T1)\}$ and $a \cdot b \cdot$`sizeof(T1)`$\leq M$.

Measurements and tuning:

- Parameters a, b, B: choose the input size N large enough, such that the matrix cannot be processed by the internal memory algorithm, and find the optimal values of a and b for each value of B mentioned above. Also, find the best B. In order to find the best values, draw plots (preferably *time per element*).
- Run the internal memory algorithms for all inputs it can handle. Again, draw plots for (*time per element*).
- Run the I/O-efficient algorithm for all inputs. Add the running time curves to the previous plots. Optionally repeat the measurements for different values of B and add them to the plots.

Write a short report that includes the figures you have plot. In your explanations the following points should be present:

- Explanations on the plots: why one algorithm is faster/slower than another, for which input sizes.
- The role of the parameter B on the performance of the algorithm.
- Does the choice of parameters a and b makes (much) difference? Why?
- Mention the name/the configuration of the computer on which you ran the experiments.

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

[CS] Siddhartha Chatterjee and Sandeep Sen. Cache-Efficient Matrix Transposition. In *HPCA 2000*, pages 195–205. `http://algo2.iti.uka.de/dementiev/courses/cache05/hpca00.pdf`.