DM207 I/O-Efficient Algorithms and Data Structures

Fall 2009

Project 1

Department of Mathematics and Computer Science
University of Southern Denmark

September 26, 2009

In this project, we study the behavior of various sorting algorithms when used in external memory. More specifically, the goal of the project is to

1. Use different I/O approaches to manipulate streams of data to/from external memory.
2. Develop an efficient algorithm for merging several streams of data to/from external memory.
3. Implement multi-way *Mergesort* in external memory.
4. Compare these implementations with the standard *Heapsort*, *Quicksort*, and (binary) *Mergesort* algorithms.

Standard sorting algorithms

As part of the project, you will need to implement a *Heap*, as well as *Heapsort*, *Quicksort*, and binary *Mergesort*. To keep the project down in size, and to increase comparability of results between different projects, we suggest you to base your implementations on the Java code in the book

> Robert Sedgewick: *Algorithms in Jave, Parts 1–4*, third edition. Addison-Wesley, 2002, ISBN 0-201-36120-5.

This code can be found online at

`www.cs.princeton.edu/~rs/Algs3.java1-4/code.txt` .

The book (and code) also exists in a C version, as well as a C++ version, which may be more useful if you choose to program your implementation in these languages.

Tasks:

1. Write a program which takes two arguments n and *filename* and creates a file named *filename* containing n random `int`s.

2. The implementation of multi-way *Mergesort* should be based on streams (input streams and output streams). Assuming your are using Java (for other programming languages, look up the equivalent library components), the following implementations should be tried:

   (a) Reading and writing is done one element at a time using unbuffered streams, as in

   ```
   in = new DataInputStream(new FileInputStream(dataFile)).
   ```

   (in C and C++, look at the `read` and `write` system calls.)

   (b) Reading and writing is done using library buffered streams, as in

   ```
   in = new DataInputStream(new BufferedInputStream(new
                FileInputStream(dataFile)));
   ```

   (in C and C++, look at the `fread` and `fwrite` functions from the `stdio` library.)

   (c) Reading and writing is done using your own buffered stream objects, where you keep a large buffer, which is read into (written from) when empty (full).

   For each of the implementations (a) and (b), as well as for (c) with various values of buffer size (including very large ones), perform the experiment of opening k streams and n times read (write) one element to (from) each of the streams. For each implementation, do this for a large n and for $k = 1, 2, 4, 8, \ldots,$ MAX, where MAX is the maximal number of streams allowed by the operating system. Try to single out a winner implementation.

3. Implement a d-way merging algorithm that given d sorted input streams creates an output stream containing the elements from the input streams in sorted order. The merging should be based on the priority queue structure *Heap*.

4. Implement a multi-way *Mergesort* algorithm for sorting `int`s. The program should take parameters n, m, and d, and should proceed by the following steps.

   (a) Read the input file and split it into $\lceil n/m \rceil$ streams, each of size $\leq m$. Each stream that is created should be sorted in internal memory using *Quicksort* before writing it to external memory.

   (b) Store the references to the $\lceil n/m \rceil$ streams in a queue (if necessary in external memory).

   (c) Repeatedly merge the d (or less) first streams in the queue and put the resulting stream at the end of the queue until only one stream remains.

5. Using the best of the stream implementations from above, make some experiment with your multi-way *Mergesort* program: try different values for n, m, and d, and identify some good choices of m and d. The test data should be random `int`s.

6. **One possible extra task (not mandatory):** Implement a version of the multi-way *Mergesort* algorithm above which tries to parallelize CPU work and I/O in the first phase by restricting the initial sorted streams to be of length $\leq m/2$, and then sorting one stream while transferring another to/from disk. For this to work, you should use threads. Compared it to your implementation above (by experiments), and take the best.

7. Implement the standard *Heapsort*, *Quicksort*, and binary *Mergesort* algorithms.

8. For various sizes of data (a few sizes in RAM and a number of sizes larger than main memory, including at least a factor 10 larger), compare the running time of your multi-way *Mergesort* algorithm, the *Heapsort* algorithm, *Quicksort* algorithm, and the binary *Mergesort* algorithm.

Formalities

Make a report of 6–8 pages describing your implementation and your experiments, on a level of detail such that others could repeat your experiments themselves. In particular, this includes reporting the compiler version, compilation options, and machine characteristics (at least disk, RAM, and cache sizes). Use plots of your experimental data (not tables), and make sure it is explained what they show. Let the y-axis be time$/n \log n$, and let the x-axis be logarithmical. Draw conclusions based on the observed data. Plots should be given as an appendix (not included in the page count above), code should be online available and an url to it should be given in the report.

You should hand in your report (in pdf) using the digital drop-box at the Blackboard page of the course (under menu item "Tools").

Deadline:

Monday, October 19, 2009, at 23:59.