DM808 I/O-Efficient Algorithms and Data Structures

Spring 2008

Project 2

Department of Mathematics and Computer Science
University of Southern Denmark

March 14, 2008

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.

The project is to be done in groups, preferably of size two.

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 require you to use the code in the book

Robert Sedgewick: *Algorithms in C, Parts 1–4*, third edition. Addison-Wesley, 1998, ISBN 0201314525.

This code can be found online at

`www.cs.princeton.edu/~rs/Algs3.c1-4/code.txt` .

Tasks:

1. Write a program which takes two arguments n and *filename* and creates a file named *filename* containing n random 32-bit integers.

2. The implementation of multi-way *Mergesort* should use the concept of *streams*. There should be two different kinds of streams: *input streams* and *output streams*. An input stream should at least support the operations `open` (open an existing stream for reading), `read_next` (read the next element from the stream), and `end_of_stream` (return `true` if the end of the stream has been reached). An output stream should support the operations `create` (create a new stream), `write` (write an element to an existing stream), and `close` (close the existing stream).

   Make implementations of streams, each using a different one of the following four I/O mechanisms. In all four cases, the actual data of the stream should be stored in a simple file.

   (a) Reading and writing is done one element at a time by the `read` and `write` system calls.

   (b) Reading and writing is done by the `fread` and `fwrite` functions from the `stdio` library. These implement their own (fixed) buffering mechanism.

   (c) Reading and writing is handled as in (a), except that now the stream is equipped with a buffer in internal memory of size B, and whenever the buffer becomes empty/full, the next B elements are read/written from/to the file.

   (d) Reading and writing is handled by mapping the file containing the stream to internal memory using `mmap` and scanning the stream as if it was an array in internal memory.

   For each of the implementations (a), (b), and (d), as well as for (c) with various values of B (including very large ones), perform the experiments 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 $k = 1, 2, 4, 8, \ldots, \textsc{Max}$, where $\textsc{Max}$ is the maximal number of streams allowed by the operating system. For each of the four stream implementations, identify their properties and limitations, and try to single out a winner.

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 32-bit integers. 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. (**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.

6. Perform experiments with the multi-way *Mergesort* program, using the best of the stream implementations from above. The data should be random 32-bit integers. Try different values for n, m, and d, and identify what are good choices of m, and d for the various sizes.

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 the best of your multi-way *Mergesort* algorithms, the *Heapsort* algorithm, *Quicksort* algorithm, and the binary *Mergesort* algorithm.

Formalities

Make a report of 5-10 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. Code should be given as an appendix (not included in the page count above).

Deadline:

Wednesday, April 16, 2008.