DM207 I/O-Efficient Algorithms and Data Structures

Fall 2011

Project 1

Department of Mathematics and Computer Science
University of Southern Denmark

September 20, 2011

Introduction:

In this project, we study the behaviour of various sorting algorithms when used in RAM and in external memory. More specifically, the goal of the project is to implement and experimentally compare

1. *Heapsort*,
2. *Quicksort*,
3. *Mergesort*, recursive version,
4. *Mergesort*, bottom-up version,
5. *Multi-way mergesort*,

and in this way get practical experiences on some of the points made during the first weeks of the course.

The project is to be done in groups of size two (although single person projects are allowed).

Standard sorting algorithms

The algorithms *Heapsort*, *Quicksort*, and binary *Mergesort* are assumed known from previous courses. To get a head start on your implementation of them, you are allowed to base your implementation on code from:

> Robert Sedgewick, Kevin Wayne: *Algorithms*, 4th Edition. Addison-Wesley, 2011, ISBN 032157351X.

This code can be found online at

`http://algs4.cs.princeton.edu/20sorting/` .

This code is written in Java, and is rather clear and instructive. It should be adjusted and tuned to include the elements discussed below. You may use any programming language for the project, and can just regard the code above as highly detailed pseudo-code.

Tasks:

In general, all the algorithms below should just be implemented on arrays of `int`'s (not more general objects, as otherwise done in the code by Sedgewick and Wayne (SW)). You should focus on optimizing CPU-time also, otherwise the effects of memory accesses inside RAM may be less pronounced. For instance, inline the small helper functions `less` and `exch` defined in the SW code, either directly, or by macros if using C or C++. If your compiler has optimization switches, ask for maximal optimization for speed of code (e.g. option `-O3` for `gcc`). Make sure you during development test that the output is sorted. Measuring algorithms not working correctly will teach us nothing.

1. Implement *Heapsort*.

2. Implement *Quicksort*. Just the plain version, not the three-way partition version also present in the SW code. You may (but are not required to) try out whether switching to Insertionsort for small instances (say, less than size 10) gives any speedup.

3. Implement (binary) *Mergesort*, recursive (i.e., top-down) version. The version in the SW code moves all elements twice during a merge, first copying from the main array to a second auxiliary array, and then merging back to the main array. This should be avoided by having the two global arrays have more identical roles, and merging while moving from the first to the second array on one level of the recursion tree, and then vice verse (merging while moving from the second to the first) on the next level of the recursion tree. You may (but are not required to) try out whether switching to Insertionsort for small instances (say, less than size 10) gives any speedup.

4. Implement (binary) *Mergesort*, bottom-up version. Again, use the two global array wiser, as described above, letting the two switch roles for each level of the merge tree.

5. Implement *Multi-way mergesort*. First implement a multi-way merge, which merges k sorted segments of one array into one sorted segment in another (similar to the binary merge used above). One merge step moves the smallest of the k front elements (of the remaining elements) in the k sorted segments. You may find this smallest element via simple linear search among the k front elements, or you may (but are not required to) try whether using a priority queue for this task (similarly to the multi-way merge in the SW code) will be faster. Again, there should be two global arrays switching roles at each level of the merge tree. You should start by sorting segments of M elements, using Quicksort, to make the initial runs.

6. Make the following experiments. In all of these, the input should just be an array of n (pseudo-)randomly generated `int`'s. Let n range from 2^{10} to around $1.5 \cdot 2^{28}$, raising it by a factor of 1.5 each step. For the multi-way mergesort, try $k = 2, 4, 8, 16, 32$ and M equal to 1/16 of the size in bytes of the L3 cache (on Linux, read `/proc/cpuinfo` to see cache size). Then boot the machine with RAM restricted to 256Mb or 512Mb (see instructions below), and do the same experiments again, except set M to 2^{13} elements. This will force a number of the input sizes to be in external memory. Note: heapsort may stall (i.e., not finish in any reasonable amount of time) even for the first size outside RAM, and the experiments may have to be truncated for that algorithm.

You should measure the running time of each sort process (for each of the values of n, k, and M above) using wall clock time (use e.g. `currentTimeMillis` in Java, or `gettimeofday` in C or C++). Be sure to test only the sorting (not e.g. creating the array of input elements, or any test that the output is sorted).

You should plot running time divided by $n \log n$ on the y-axis, and $\log n$ on the x-axis. For the memory constrained experiments, you may want to plot log(running time divided by $n \log n$) on the y-axis, since the y-values will differ more.

Formalities

Make a report of 6–8 pages describing your implementation (and its relation to any SW code it is based on) 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 (such as disk, RAM, and cache sizes). Use plots of your experimental data (not tables), and make sure it is explained what they show. 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 (not included in the report) and an url to it should be given in the report.

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

For groups of size more than one: For formal reasons, you will need to designate who wrote which part of the programs and report.

The project will be evaluated by pass/fail grading. The grading will be based on:

- The clarity of the writing and of the structure of the report.
- The thoroughness of the experiments—execution as well as discussion.
- The amount of work done.

Deadline:

Monday, October 17, 2011, at 23:59.

Booting with less RAM

The following can be done on machines in the terminal room (and probably any Ubuntu Linux installation). It will make the operating system only see the specified amount of RAM. In the terminal room, don't use any of the nine machines meant for external logon (see handed out diagram of the terminal room), and don't use any machine with a running batch job from another user (check first few lines of output of the `top` command).

1. Push briefly the physical power button on the machine.

2. Choose “Shut down” from the appearing menu on the screen, and wait for the shut-down to complete.

3. Then press press the power button again to start up the machine. Immediately after, hold down the Shift key. Keep it down until the ASCII-based GRUB menu appears after some time.

4. With first line of the menu highlighted, press `e` for edit, move (with arrow keys) to line starting with `linux`, and move to the end of this line with Ctrl-e.

5. Write (append) the string ” `mem=256M`” to end of line. (The character ’`=`’ will be on the key just left of the backspace key.)

6. Press Ctrl-x, and wait for the boot to complete.

7. Perform the experiments.

8. Shut down and start the machine again, this time with no editing of the GRUB menu. This will make the machine return to its default settings.