Algorithms in Secondary Memory

Project Assignment

2012-2013

Assignment

In this assignment you are asked to implement an external-memory merge-sort algorithm (such as the one described in Section 15.4.1 of TCB), and examine its performance under different parameters. As a warm-up exercise you will need to explore several different ways to read data from, and write data to secondary memory. The overall goal of the assignment is to get real-world experience with the performance of external-memory algorithms.

Follow the outline of tasks below. You need to document your findings in a report that needs to be handed in together with your implementation. Code may be written in either C++ or Java.

1 Project outline

1.1 Reading and writing: streams

Your merge sort implementation will need to read data from, and write data to disk. A first step, therefore, is to develop *stream* classes that can sequentially read and write a file consisting of 32-bit integers. You will need two types of streams: *input streams* and *output streams*. The input stream should support the following operations: `open` (open an existing file for reading), `read_next`, (read the next element from the stream), and `end_of_stream` (a boolean operation that returns true if the end of stream has been reached). The output stream should support the following operations: `create` (create a new file), `write` (write an element to the stream), and `close` (close the stream).

Use the following four distinct I/O mechanisms to obtain four different implementations of the input and output streams.

1. Read and write one element at a time using the `read` and `write` system calls. These calls are available in C/C++ through the `io.h` header (if you're programming on windows in Visual C++) or through the `unistd.h` header (on Unix/Linux). If you are doing the project in Java these system calls can be mimicked. You can mimic `read` by calling `readInt()` on a `java.io.DataInputStream` that is wrapped directly around a `java.io.FileInputStream`, a illustrated by the following code snippet:

```
InputStream is = new FileInputStream( new File("input.data" ) );
DataInputStream ds = new DataInputStream(is);
ds.readInt();
```

The `write` system call can be mimicked similarly.

2. Read and write using `fread` and `fwrite` functions from the C `stdio` library which implements its own buffering mechanism for these functions. If you are doing the project in Java these functions can be mimicked. You can mimic `fread` by calling `readInt()` on a *java.io.DataInputStream* that is wrapped around a *java.io.BufferedInputStream* that itself is wrapped around a *java.io.FileInputStream*, a illustrated by the following code snippet:

   ```
   InputStream is = FileInputStream( new File("input.data" ) );
   BufferedInputStream bis = new BufferedInputStream( is );
   DataInputStream ds = new DataInputStream( bis );
   ds.readInt();
   ```

   The `fwrite` function can be mimicked similarly.

3. Read and write are implemented as in step (1), except that now you equip your streams with a buffer of size B in internal memory. Whenever the buffer becomes empty/full the next B elements are read/written from/to the file.

4. Read and write is performed by mapping and unmapping a B element portion of the file into internal memory through *memory mapping*. Whenever you need to read/write outside of the mapped portion, the next B element portion of the file is mapped.

   Memory mapping is available in C/C++ through the mmap/munmap functions of the `<sys/mman.h>` header (in Linux) and the `CreateFileMapping` function of `<windows.h>` (in Windows). Alternatively, the `Boost` C++ library[1] provides platform-independent interfaces to memory-mapped files.[2] In Java, you should use the `map` method of the `java.nio.channels.FileChannel` class.

   For this particular implementation you are required to do a little research on what *memory mapping* actually is; you are expected to explain this concept in your report.

Experiment with each stream implementation and determine which implementation is the most performant. The goal is to determine which works the best, and to clearly document the process of this discovery in the report. What are the limitations and advantages of each implementation? In particular:

- Try opening k streams (on distinct files) and read/write one element to/from each stream N times. Try for large N and $k = 1, 2, \ldots, \textsc{Max}$ where Max is the maximum number of streams you are able to open.

- For implementations (3) and (4), experiment with different values of B (including very large ones). Identify the optimal values.

Don't forget to describe, in your report, the environment on which your experiments have been performed (machine type, hard disk type, operating system, total memory available, programming language used, and libraries used).

[1] `http://www.boost.org/`

[2] It actually provides 2 such interfaces, one in the `iostreams` sublibrary, and one in the `interprocess` sublibrary.

1.2 Multi-way merge

Implement a d-way merging algorithm that, given d sorted input streams of 32-bit integers, creates a single output stream containing the elements from the input stream in sorted order. The merging should use a priority queue (e.g., a heap) to obtain the next element to be output at all times.

1.3 External multi-way merge-sort

Implement an external memory multi-way merge-sort algorithm for sorting 32-bit integers. The program should take as input an input file and the following parameters:

- M — the size of the main memory available, in number of 32-bit integers;
- d — the number of streams to merge in one pass;

and sort as follows. Let N be the size of the input file, measured in number of 32-bit integers.

1. Read the (single) input file and split it into $\lceil N/M \rceil$ streams, each of size at most M. Each stream that is created should be sorted in internal memory using heapsort, (internal-memory) mergesort, or a sorting method that has comparable $O(n \log n)$ worst-case performance before being written to disk. In particular, you can reuse the sort algorithm provided with the language you use if it meets the performance requirement.
2. Store the references to the $\lceil N/M \rceil$ streams in a queue (if necessary in external memory).
3. Repeatedly until only one stream remains, merge the d first streams in the queue, and put the resulting stream at the end of the queue (if only $x < d$ streams remain in the queue, merge those).

Experiment with your merge-sort implementation using the best stream implementation from 1.1. Hereto, you should generate (unsorted) input files that contain randomly generated 32-bit integers. Try different values of N, M, and d. Identify what are good choices of these parameters for various input sizes (including very large ones). In case that the input file is small enough to fit in memory, compare your merge-sort implementation with a main-memory sort (e.g., heapsort, mergesort). Document everything you discover in your report.

2 Tips

- Run your programs a suitable number of times for each input size and use the average running time. This should level out fluctuations due to other processes.
- There are quite a number of Unix commands to time the execution of the program. Among these are time and timex. In some shells (e.g. csh), there are also built-in version of time. There are also corresponding unix routines called `times`, `getrusage`, and `gettimeofday`. Their availability may differ from machine to machine. The Windows API also provides `QueryPerformanceCounter` and `QueryFrequencyCounter` that can also be used to time the execution. Both the Boost library and the C++11 STL also provides a `chrono` class that provides timing in a platform independent way.

- When comparing different algorithms (or the same algorithm with different parameters), make sure to run them on the *same input file* in order to get meaningful results.

3 Modalities

The assignment has the following modalities:

1. This project assignment contributes 6/20 to the overall grade; the written exam contributes the remaining 14/20 points.

2. The assignment will be graded on (1) the implementation itself and (2) a report that you need to write describing both the implementation and the experimental work.

3. The assignment should be solved in groups of 2 (if we have an odd number, one group of 3 students will be allowed). You are asked to send, per group, the names of the group members to Mr. Francois Picalausa (`fpicalau@ulb.ac.be`) by October 22 at the latest. If you cannot find a partner, please indicate so by sending an email to Mr. Picalausa, who will hook you up with a partner.

4. This assignment is mandatory. If you do not make the assignment, you cannot pass the course in the first exam session.

5. You will have to create a mercurial repository[3] in the `INFO-H-417` repository group at `http://wit-projects.ulb.ac.be/rhodecode/` to submit both your report and your code. The username and password to login to this system correspond to your ULB/VUB NetID. The repository will be named `project-<student1>-<student2>`, where `student` corresponds to your username. It is recommended that you create this repository *as soon as possible* to avoid last minute technical difficulties, and that you use it throughout the project to synchronize your changes.

6. Your solution should be pushed to the repository *no later than 22 December 2012*. You get a penalty of -1/6 points for each day that your solution is delayed. Only the latest commit will be considered as the solution.

7. As stated above, the project will need to use a *priority queue* and implement *heapsort* or equivalent main-memory sorting algorithm. You are only allowed to use existing existing code and existing libraries for these two aspects; all other code must be written by you for the project, unless explicitly stated otherwise in the assignment above. For example, you can use the `std::priority_queue` class from the C++ STL to implement priority queue (if you do the project in C++), or the `Java java.util.PriorityQueue` (if you do it in `Java`).

8. Sharing of code between groups is not allowed. (Groups may, however, verbally discuss ideas on how to tackle the project).

9. Plagiarism, in the sense of copy-pasting from existing reports or copy-pasting existing external memory merge-sort code is not allowed. In case plagiarism is detected, students risk being punished according to article 34 of the exam regulations shown in Figure 1.

[3] `http://mercurial.selenic.com/wiki/Tutorial`

Art.34 In case of fraud or plagiarism during an examination or during a test at an interim date during the academic year, or in relation with the preparation of written reports or papers, the course professor reports the case in writing prior to the jury deliberation to the relevant academic authority levels for disciplinary matters. A copy of that fraud report is addressed to the jury chairmen. The student can ask to be heard by a jury chairperson prior to the jury deliberation, in presence of the related course professor. Without prejudice to the disciplinary processes at the University Faculty level, in case of fraud the student points for the related course are brought down to 0/20. The jury further can:

- *decide to cancel the examination session;*
- *decide to refuse the student access to both examination sessions of that academic year.*

Figure 1: Excerpt of the Exam Regulations concerning fraud.