CSCD320 Project, Winter 2012, Eastern Washington University. Cheney, Washington.

Name: Eric Fode EWU ID:00530214

Abstract

The solution that I have chosen to use to solve the problem of sorting huge data sets is well tested, and has been used in many other instances. The code in this project was based off of another implementation of external merge sort from here http://www.codeodor.com/index.cfm/2007/5/14/Re-Sorting-really-BIG-files—the-Java-source-code/1208 and then modified to work on a newline delimited list instead of a csv, and the functionality to copy rows of data along with the sorted items was removed. I also tweaked the number of item read in from 10k to 10m because single ints are much smaller then rows of ints, and the larger the sets that we can sort in RAM the better. As well as implementing this algorithm I took some time to look at ways that it could be improved to work faster.

The Algorithm

The algorithm used is called an external merge sort. The basic idea behind it is to sort as much of the file as you can at once in RAM and then save it into a ”chuck” file. Then after the whole file has been chuncked the chunk files are merged together using the same algorithm that is used in a traditional merge sort but with the added complication of maintaining multiple lists of numbers to be merged as opposed to just two (as in a traditional merge sort) and the complication of only having small chunks of each chunk RAM at a time.

The only part of this that bears elaborating is the process of merging multiple files at the same time. This is done by keeping the first int in each file buffer and continually taking the smallest of them. Then removing it from the file buffer it belongs to refilling the file buffer with the next int in the chuck and then repeating the whole process with all of the file buffers again. This repeats until there are no items left in any of the files being merged.

Implementation Details

The implementation of my external merge sort was based off of the site that was listed above. Major changes include:

1. The compatibility with full tables that had multiple columns that could be sorted on and were preserved throughout the merge was removed.
2. Because of the previous simplification there were many variables simplified from arrays of types to simple types, some functions removed, and header information no longer is accounted for.
3. Instead of a custom merge sort (or quicksort) being used to do the sorting I chose to use the built in sort in Java. This was done for the following reasons: 1. It would be expected that the built in sort would be near optimal 2. It is expected that it is optimized for the JVM 3. As a result of the previous two points I believe that it is much more effective to use the built in sort. Also this does not violate the terms of the assignment because the point was to solve the challenge of sorting a massive file, this call to sort is only doing a small trivial part of the work.

I also took the time to improve the grammar used in the code.

Name: Eric Fode **EWU ID:00530214**

Performance/Results

I discovered that this is a fairly bad implementation of external merge sort. This is for a few reason that identified.

1. To many type conversions. It would have been much faster if the file was binary instead of human readable.

2. File IO operations happening one item at at time. This was a massive problem, it instead a few thousand lines were read in (or wrote out) at a time it would have performed much better. This is not quite as simple as it sounds though. because we were using a human readable file there was no guarantees about data length consequently you can't just jump around in the file and expect to end up the boundary of anything.

3. Single threaded. This application would have performed much better if not only had the information been read in in chunk but the type conversions and then sorting of the chunks would have taken place in separate threads so that IO was not blocked and and there was less idle time. This would probably only have been useful if one of two condition were fulfilled though. 1. The threads sheared memory so that the only communication necessary was notifications of the disk being ready to write to, the sorting thread finishing a chunk and the reading thread finishing a chunk. 2. If the chunks were extremely small and as a result could be communicated and each thread operation would be unlikely to block each other. Out of these two possible optimizations using threading I would go for the second one but make the chunk sizes large enough that IO was not slowed down.

Future work

This would include implementing a way to do chunked IO, and implementing threading.