A Comparison of Sorting Methods for Sorting Data on Lower End Hardware and Software 2nd revision

Dan Cojocaru
Department of Computer Science,
West University of Timișoara
email: dan.cojocaru00@e-uvt.ro

Abstract

As smartphones become more affordable, people in poor countries start to get connected to the internet. The speed of the connection and the performance of the devices, however, can negatively impact the experience.

In this paper, I analysed the performance of sorting algorithms when applied to streams of data coming through slow connections and determined that splitting the data into smaller chunks and handling those can significantly improve the performance.

Contents

1 Introduction **3**
1.1 Motivation 3
1.2 Short description of findings 3
1.3 Contributions 3
1.4 Reading instructions 4

2 Approach **4**
2.1 Implementation 5

3 Results **8**

4 Related Work **9**

5 Conclusions and Future Work **10**

1 Introduction

1.1 Motivation

More of the world is getting connected to the internet, especially in developing countries, but also in remote areas of developed nations. This provides a challenge, as both the hardware and the software of these new internet users are significantly lower in performance than what most of the already connected world uses.

Since most of the world is used to potential internet speeds up to (and beyond) 100 Mbps, developers tend to assume that the bottleneck of an algorithm would be how quick a processor can process their data, rather than obtaining the data in the first place.

Given this (and recent internet connections), I decided to write this paper analysing how a different approach focused on optimising algorithms based on the speed of data throughput rather than data processing can be an important point of focus.

1.2 Short description of findings

Although initially I thought about exploiting the only commonly known online sorting algorithm[1], insertion sort (as described in [Aigner and West, 1987]), the results were disappointingly uninteresting.

I then turned to another approach. I started looking into chunking the incoming input and sorting little chunks at a time. Using this approach, I observed significant improvements for these following algorithms I analysed:

- Quicksort [Hoare, 1962]
- Heap sort
- Merge sort [Pardo, 1977]
- Bubble sort [Astrachan, 2003]
- Selection sort

1.3 Contributions

Instead of taking the usual approach of comparing sorting algorithms in a perfect scenario with no external influence, in this paper I compare sorting algorithms in real world situations where things like slow internet connection have a bigger impact than which algorithm is faster.

[1]online algorithm = an algorithm which can process data as it is available rather than waiting for the whole dataset before starting to process it

1.4 Reading instructions

Since this paper is more focused on the practical experiments and results, the theoretical side will not be focused much on.

In the Approach [2] section, the details about the implementation will be described, including technical details which will help the reader in creating their own implementation.

In the Results [3] section, the results gathered through my implementation will be discussed, including potential improvements or different approaches that can yield different results.

In the Conclusions and Future Work [5] section, this paper will be summarised, and possible further avenues of research will be discussed.

2 Approach

In order to gather the data for this paper, I decided to write a server that emulated bad internet connections and a client which communicates to it.

The client will establish a connection to the server and ask it to start a *sorting session*. A sorting session is characterised by the following steps:

- The server will record the time at which the session is started.
- The server will communicate one by one each item in the list to be sorted. The communication of each item will have a delay that the client asked for between them.
- The server will ask the client to sort the received list.
- The client will send the sorted list back.
- The server will record the time at which the session is over, will check the received list to be correctly sorted and then will reply to the client informing it of the time it took for the list to be sorted.

This approach will impose several delays in the communication of the list. Even though the client may ask for a 0 millisecond delay, the TCP communication socket[2] will ensure that the communication is *near*-instantaneous, but **not** instantaneous. Furthermore, in my implementation of the client, a GUI[3] is maintained in order to communicate progress of sorting, the update of said GUI also taking time. Both the non-instantaneous communication and communicating progress are expected real world behaviours of user-facing applications and are therefore justifiably included in the time needed for a sorting session to complete.

[2] (in networking context) socket = a bidirectional channel with which a client and a server can send each other data

[3] GUI = graphical user interface

2.1 Implementation

I implemented my approach using the Java programming language, using TCP for communication between the server application and the client application.

The server is a console application which instantiates a TCP server and waits for a client to connect and communicate.

The client is a JSwing application with a GUI where a user can select the method to test. After selection, the client connects to the server and then communicates in order to benchmark the sorting method.

I devised a binary message-based protocol between the client and the server in order to make the communication as efficient as possible. In this protocol, a message starts with its length in bytes, and then the contents of the message follow. Furthermore, the messages are structured: each message is a structure which contains multiple ordered fields. Different structures can be differentiated by fields which contain a constant value. In my implementation, the first field of each structured is named TAG, and it contains a 32 bit integer constant identifying each structure. This protocol allows for optional fields at the end of a structure, since the length of the message which is first sent allows the program to inspect when the structure ends as it progresses through it. By convention, the name of the fields that are optional end with a question mark in the spec of the protocol I designed.

Here is the spec I created for my protocol:

```
== TYPES ==
TAG = i32

MILLISECONDS = i64
SECONDS = i64
EPOCH = i64

STRING =
    CHARACTER_COUNT = i32
    CHARACTERS = [i8] of size CHARACTER_COUNT

COMPOSED_STRING =
    STRING_COUNT = i32
    STRINGS = [STRING] of size STRING_COUNT

PING =
    TAG = TAG(-1)
        expects PONG

PONG =
    TAG = TAG(-2)

== CLIENT MESSAGE TYPES ==
OK =
    TAG = TAG(0)

ERROR =
    TAG = TAG(1)
    WHY? = STRING assumed STRING(0, [])

START_SORT =
    TAG = TAG(2)
    DESIRED_ELEMENT_COUNT = i32
    DESIRED_ELEMENT_SEND_DELAY? = MILLISECONDS assumed MILLISECONDS(0)
    MINIMUM_DESIRED_VALUE? = i32 assumed i32(-2147483648)
    MAXIMUM_DESIRED_VALUE? = i32 assumed i32(2147483647)
        expects SORT_START_SENDING_ELEMENTS

FINISHED_SORT =
    TAG = TAG(3)
        expects SEND_FINISHED_SORT_RESULTS

FINISHED_SORT_RESULTS =
    TAG = TAG(4)
    ELEMENT_COUNT = i32
    SORTED_ELEMENTS = [i32] of size ELEMENT_COUNT
        expects SUBMISSION_ACCEPTED

LONG_OPERATION =
    TAG = TAG(2000)
    OPERATION = TAG
    EXPECTED_DURATION = SECONDS
        expects LONG_OPERATION_APPROVAL

== SERVER MESSAGE TYPES ==
OK =
    TAG = TAG(0)

ERROR =
    TAG = TAG(1)
    WHY? = STRING assumed STRING(0, [])

SORT_START_SENDING_ELEMENTS =
    TAG = TAG(2)

SORT_ELEMENT =
    TAG = TAG(3)
    ELEMENT = i32

SORT_FINISHED_SENDING_ELEMENTS =
    TAG = TAG(4)
    ELEMENT_COUNT = i32
        expects OK

SEND_FINISHED_SORT_RESULTS =
    TAG = TAG(5)
        expects FINISHED_SORT_RESULTS

SUBMISSION_ACCEPTED =
    TAG = TAG(6)
    SORTED_CORRECTLY = bool
    TIME_ELAPSED = MILLISECONDS
    SUBMISSION_ID = STRING
        expects OK

LONG_OPERATION_APPROVAL =
    TAG = TAG(2000)
    WILL_TIMEOUT_AT = EPOCH
```

Here is an example of a normal client - server interaction:

```
CLIENT            - creates a connection
CLIENT >>> SERVER - START_SORT(2, 100, 10)
           SERVER - Saves the current time
CLIENT <<< SERVER - SORT_START_SENDING_ELEMENTS
CLIENT <<< SERVER - SORT_ELEMENT(<element>) repeatedly
CLIENT <<< SERVER - SORT_FINISHED_SENDING_ELEMENTS(4, <count>)
CLIENT >>> SERVER - OK(0)
CLIENT            - sorts the elements
CLIENT >>> SERVER - FINISHED_SORT(3)
CLIENT <<< SERVER - SEND_FINISHED_SORT_RESULTS(5)
CLIENT >>> SERVER - FINISHED_SORT_RESULTS(
                        4,
                        <count>,
                        [<sorted_elements>])
           SERVER - measures the time elapsed while sorting
CLIENT <<< SERVER - SUBMISSION_ACCEPTED(
                        6,
                        true,
                        <time for sort>,
                        <identifier - unused in a simple server>)
CLIENT >>> SERVER - OK(0)
CLIENT            - ends the connection
```

3 Results

Initially, I expected insertion sort, which is an online algorithms, to perform better in an online context. Even so, my expectations were not met when I put them to text, insertion sort performing worse than other algorithms on average. This is despite other algorithms having to wait for the data to fully arrive before starting to process it.

I then tried different methods of splitting the received data into smaller chunks that can be processed as they are ready, instead of using all the data as one big chunk. This approach yielded surprising results.

The attempted methods of dividing the work into chunks were:

- Equal Chunks – after processing a chunk, wait until the amount of unprocessed data equals the amount of processed data before reprocessing
- Constant Chunks – a constant number of unprocessed items will be taken as a chunk and processed each time
- Exponential Chunks – a counter starting at 1 is kept, and each time the amount of unprocessed items is greater or equal than the counter, the items are processed as a chunk, and then the counter is multiplied by a specified value

The processing of a chunk implies sorting it using a given sorting algorithm and then merging it with the existing processed data.

Although in my experimentation, which included constant chunks of size 500 and 2000 as well as exponential chunks with a factor of 3 and 5, a conclusive result was not reached on what values work better, further experimentation being needed for this, results constantly show that dividing the received data into chunks yields significantly better results.

On a test dataset of 10,000 randomly generated numbers with a 20 ms delay between sending each item, the results were as follows:

- Insertion sort: 3 minutes, 31 seconds, 210 milliseconds
- Quicksort: 3 minutes, 29 seconds, 736 milliseconds
- Heap sort: 3 minutes, 31 seconds, 234 milliseconds
- Constant Chunks(500) Quicksort: 3 minutes, 28 seconds, 739 milliseconds
- Constant Chunks(500) Heap sort: 3 minutes, 30 seconds, 656 milliseconds
- Constant Chunks(2000) Quicksort: 3 minutes, 29 seconds, 130 milliseconds
- Constant Chunks(2000) Heap sort: 3 minutes, 30 seconds, 739 milliseconds
- Exponential Chunks(3) Quicksort: 3 minutes, 28 seconds, 960 milliseconds
- Exponential Chunks(3) Heap sort: 3 minutes, 30 seconds, 729 milliseconds

- **Exponential Chunks(5) Quicksort**: 3 minutes, 26 seconds, 772 milliseconds
- Exponential Chunks(5) Heap sort: 3 minutes, 30 seconds, 479 milliseconds

Significant improvement start to appear beginning with the 25,000 items to sort mark, where a difference of up to 20 seconds can be observed:

Chunking method	Sorting algorithm	Time for sorting 20,000 items
No chunking	Insertion sort	38 seconds 423 milliseconds
No chunking	Quicksort	41 seconds 172 milliseconds
Equal chunking	Quicksort	42 seconds 659 milliseconds
Constant chunking 500	Quicksort	27 seconds 962 milliseconds
Constant chunking 2000	Quicksort	23 seconds 843 milliseconds
Exponential chunking 3	Quicksort	24 seconds 484 milliseconds
Exponential chunking 5	Quicksort	25 seconds 559 milliseconds

4 Related Work

There are many works related to comparing sorting algorithms, each with it's own specifics and advantages. Some of them are:

- Zuluaga, M., Milder, P., Püschel, M. (2012, June). Computer generation of streaming sorting networks. In DAC Design Automation Conference 2012 (pp. 1241-1249). IEEE.

  This paper discusses a hardware approach to a new method of sorting using a new hardware names "streaming sorting networks". My solution is about using current hardware and optimising the software.

- Kocher, G., Agrawal, N. (2014). Analysis and Review of Sorting Algorithms. IJSER, 2(3).

  This paper discusses 6 sorting algorithms and compares them. This can be a good aid in choosing a sorting algorithm to combine with my described methods in this paper. The 6 sorting algorithms are:

  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Quick Sort
  - Merge Sort
  - Heap Sort

- Alkharabsheh, Khalid & Alturani, Ibrahim & AlTurani, Abdallah & Zanoon, Dr.Nabeel. (2013). Review on Sorting Algorithms A Comparative Study. International Journal of Computer Science and Security (IJCSS)

- Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., & Zagha, M. (1991, June). A comparison of sorting algorithms for the connection machine CM-2. In Proceedings of the third annual ACM symposium on Parallel algorithms and architectures (pp. 3-16).

  This paper describes 3 sorting algorithms implemented in highly optimal and parallel manners on the Connection Machine CM-2 supercomputer. While it may seem unrelated to the world of using normal computers and comparing relatively small sets of data, some lessons can still be learnt.

- Bharadwaj, A., & Mishra, S. (2013). Comparison of Sorting Algorithms based on Input Sequences. International Journal of Computer Applications, 78(14).

  This paper discusses 5 sorting algorithms and compares them. The algorithms are:

  - Insertion Sort
  - Bubble Sort
  - Selection Sort
  - Merge Sort
  - Index Sort

5 Conclusions and Future Work

To summarise, in this paper I analysed different approaches to sorting data on devices with low internet speed and processing power by imposing a delay on the receiving of data as well as sending each item to be sorted one by one, enabling optimisations based on having the input partially available.

I discovered that, despite being an *online algorithm*, insertion sort does not perform to expectations.

Instead, a combination of traditionally fast sorting algorithms combined with splitting the input into smaller chunks and processing them as they are available proved to be a better solution.

The findings of this paper can be further improved by testing for more possible methods of chunking the input, which could yield different results - better or worse.

Another area in which this paper can be expanded upon is by considering negative effects by using this approach without delays, assuming a very quick internet connection and very fast processing time, seeing if performance is greatly reduced compared to the traditional approach of sorting the entire list or stream at once.

Another important topic that can be addressed is the importance of the pivot choice in Quick Sort, as inconsistent results in its performance in this paper may be due to a randomly chosen pivot being chosen at times as a badly performing one.

References

[Aigner and West, 1987] Aigner, M. and West, D. B. (1987). Sorting by insertion of leading elements. *Journal of Combinatorial Theory, Series A*, 45(2):306–309.

[Astrachan, 2003] Astrachan, O. (2003). Bubble sort: an archaeological algorithmic analysis. *ACM Sigcse Bulletin*, 35(1):1–5.

[Hoare, 1962] Hoare, C. A. (1962). Quicksort. *The Computer Journal*, 5(1):10–16.

[Pardo, 1977] Pardo, L. T. (1977). Stable sorting and merging with optimal space and time bounds. *SIAM Journal on Computing*, 6(2):351–372.