Does multithreading speed up webcrawlers?

Simon Klaver
Leiden University
simonrklaver@gmail.com

ABSTRACT

Not as far as this project is concerned.

Keywords

Webcrawling

1. INTRODUCTION

Crawling the web is something which search engines do often, and which can take a lot of time depending on how the content is crawled, and how it is stored. This means that while crawling webpages is in itself an easy task, it depends on multiple factors how fast it is. These factors include, but are not limited to: the parser, the indexing method, and the overall complexity of the code. For example Google, which is the part of Alphabet which harbours one of the largest search engines on the world wide web, does a lot of crawling and indexing to supply the results for any search query within a fraction of a second. To achieve this, a highly optimised crawler has to be designed and used. To achieve higher speeds, multithreading most likely plays a major role.

2. FRAMEWORK

2.1 Initial code

For this project a sequential implementation of a webcrawler is given (see the 'GIVEN.zip' file in the SOURCE folder). This includes an html parser, an indexing method, and both a crawler (webspider) and a search engine (webquery). It can be made by executing 'make', and executed by executing 'make run'. The code is written by Simon Klaver, who is also author of this paper.

2.2 External libraries

To enhance the code, the external library TCLAP is used. This software can be downloaded from
`http://tclap.sourceforge.net/`, and is already given in the zip containing this report.

ACM ISBN 0.
DOI: 0

2.3 Compiling and running the code

After extracting the zip file, go to the folder SOURCE, extract 'SOURCE.zip' and open a terminal prompt in the folder C++. Then, to compile the code, execute the command 'make'. To run the code, either run 'make run' or 'make debug' (the latter is less error-prone, but creates more output).
Be aware that when running 'make clean', to clean up the folder of the executable file and the object files, the folders containing the indexes will also be deleted by default. To change this, either alter the Makefile, or simply copy the folders out of the current one, run the cleaning command, and copy them back in.

3. CODE ENHANCEMENT

To enable multithreading in the given code, first of all the Webspider class should be parallelised, which means a superclass has to be written which will execute the webspider in different threads. For this, the class Threadspider is written, which spawns all threads and joins them with the main thread afterwards. Webspider then is optimised to use a deque for the queue, so multiple threads can just take the front element in the queue and delete it, instead of having a pointer to the current element which is shared by all the threads. In theory there should be no difference, but the unordered_set which holds the queue in the given code is unordered as the name implies, so when adding new elements it is not given that they are inserted after the current element, and therefore that when incrementing the pointer the program will reach a new element instead of an old one or the end of the queue. Aside from that, mutexes have been placed around code involving the queue and code involving indexing to prevent errors regarding multiple threads accessing the same element in the queue, or multiple threads accessing the same file at the same time.
At some point however, the original parser HTMLSTREAMPARSER broke due to something, and was unusable even though nothing in the original code regarding it seemed changed. So a new html parser had to be found. After searching and having found both `libXML` and `Gumbo`, both of which lacked usable examples and overall usability as the linking did not work, a first-party parser was used, and implemented in class Htmlparser. This code is entirely self-written.
To double down on that: Code which is not self-written has in comments above said code a mention of who else wrote it.

4. RESULTS

The results are generated by running 'time make run', after which the total execution time is taken. Alternatively the user time could be registered, but is shows the same pattern as the total execution time. The results can be found in Figure 1.

The non-scaling aspect of the results probably mean that the parsing of the sites, which is about the only thing which is multithreaded in this project, is done so quickly by the program that it does not matter whether it runs in one or more threads. Therefore the bottleneck should either lie at the downloading of the web pages, or the indexing, both of which cannot be multithreaded: The first one is because libcurl does not like the std::thread module for some reason, and most crashes are circumvented by placing a std::mutex on the downloading, and the second because files tend to get erroneous when multiple threads are writing to it at the same time.

Amount of threads:	1	2	4	8
	22:45	22:36	23:08	22:11

Figure 1: Total execution time of the parser with given amount of threads

5. CONCLUSION AND DISCUSSION

The given code cannot be optimised by implementing threading, as adding even up to 8 times as many threads does not show a significant reduction in total execution time, or even any reduction at all when looking at the result of using 4 threads. Aside from that, the code also crashes due to segmentation faults or halts at random in a lot of cases, which suggest the implementation isn't that great. However, due to time constraints, this issue cannot be investigated further.