Performance Analysis of Web Crawler System

Hongli Bu

December 8, 2018

1 Description

1.1 Background

A web crawler (also known as a web spider or web robot) is a program which browses the World Wide Web in a methodical, automated manner. Web crawlers are mainly used to create a copy of all the visited pages, validating HTML code or gathering specific types of information from Web pages.
In my program, I built a crawler to gether question tags from Stackoverflow. When question ID is provided, the crawler can extract all tags related to the content of the question ID. Of course, the crawler can gather other information. But, as my goal is just to evaluate the parallel model, it is unnecessary to extract too much information.

1.2 Sequential Version

In **crawler_seq.go** file writes the sequential version of the web crawler.The basic idea of the sequential model is that every url(one line in the input file) will be read and then parsed sequentially through the net/http package of Golang. More specifically, in the parsing modular, I traverse every html element, detect $< a >< /a >$ tags and extract the href attribute of each ¡a¿ tag. If the value of the href attribute matches the prefix of question tag link pattern, I will put it into the result map.Finally, the result map will return.

1.3 Parallel Version

Instead of doing all tasks in the main thread, I designed a parallel model(**crawler_para.go**) for the web crawler system. In the program, I defined goContext structure, where two channels are stored. The tag channel is used for tag string lists which are sent by work goroutines after finishing parsing procesure and received by main thread to form the result tag map. The question channel is used for question urls which are sent by main thread after reading from input file and received by goroutines to carry out parsing procesure.
The whole workflow is like this: Firstly, the size of goroutines is set and worker function is defined. Then, the main thread reads urls from the input file and temporarily stores them in a string list called block. When the block size equals to the upbound(can be set to any positive integer values), it will be sent to the question channel. So at this time, work threads weak up and parse the urls in the block. As one thread finish its task, it will send tags it found to the tag channel. At this time, main thread ithat has been blocked waiting wakes up to process tag lists, and finally form the result tag map.
Meanwhile, I used close() function for bothe of channels. When the main thread finish reading, it close the question channel, and when all the worker goroutines finish parsing, the last worker close the tag channel.

1.4 Hardware

In order to run timing measurements on both sequential version and parallel version, an iMac with Intel Core i5 processor and 8GB memory is used. Meanwhile, the processor speed is 2.9GHz and the total number of cores is 4.

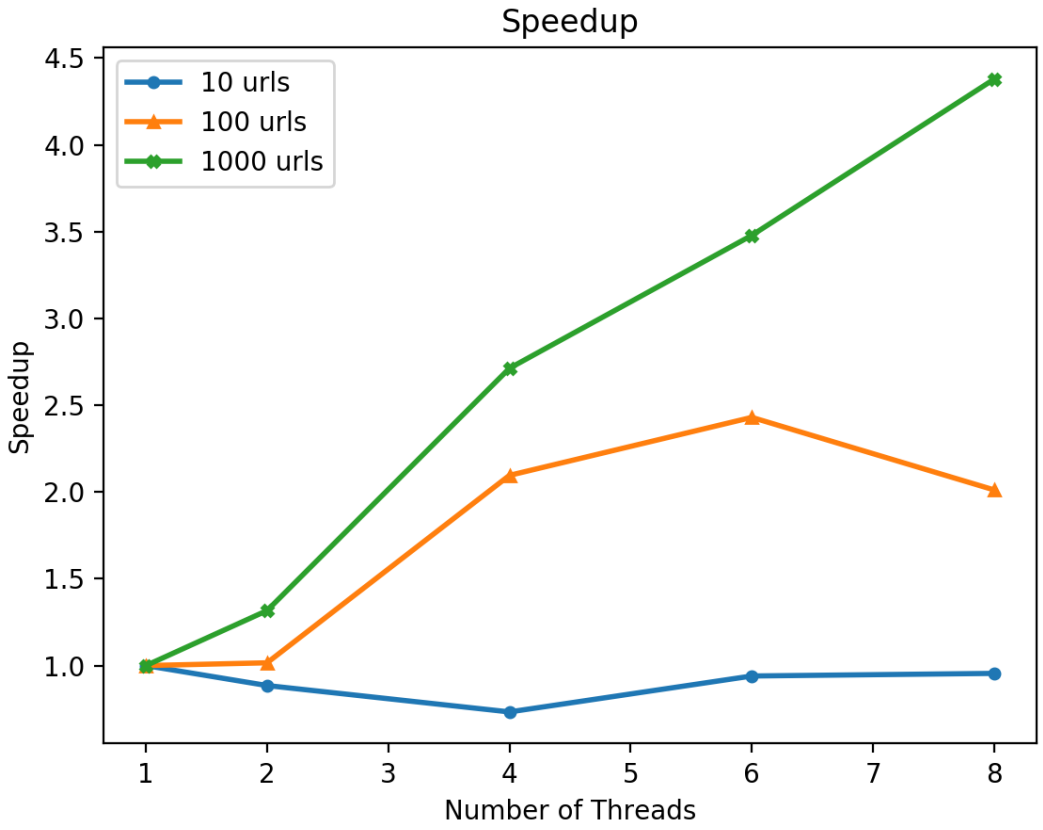

Figure 1: Speedup with different file sizes

2 Experiment

2.1 Configuration

To test speedup and reflect granularity, I chose three input files with different sizes, they include ten, one hundred, one thousand of question ids respectively. Here, we set url block to be 20, that is, every thread will process 20 urls sequentially every time it is woken up.

3 Analysis

We can see the speedup result in Figure 1. Firstly, when the input file has 10urls(less than block size), the speedup is between 0.8 0.9. It makes sense, because actually, when an input file is less than the block size, only one worker thread works. Moreover, as the lock/unlock processure and channel blocking are time-wasting, the system slows down. This is a example of coarse-grained synchronization. As the file increasing, fine-grained synchronization achieved. As the picture shown, when input files come to 100 url and 1000 urls, speedup increased dynamically except when thread number equals two. This is because at this situation, only one worker goroutine is sprawned. Although, I/O and computation can do in parallel, the whole computation is done byonly one worker thread. When the number of thread equals 4, 6, 100 urls and 1000 urls can be processed in a fine-grained synchronization without communication overhead. When thread equals to 8 and the input file have 100 urls, I think synchronization overhead may occurr and lead to a relative smaller speedup.

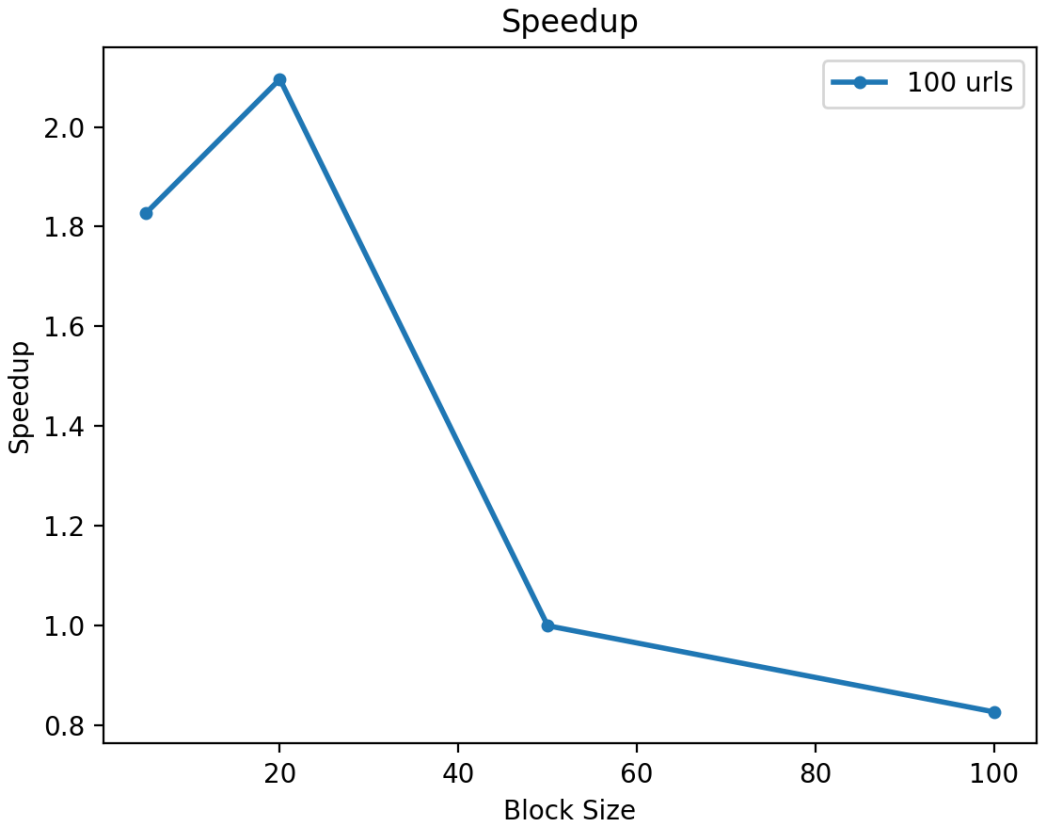

Figure 2: Speedup with different block sizes

4 Envaluation

4.1 Hotsopts and Bottlenecks

The hotspots I think are the computation parts, in which url parsing is conducted. Every url needs to enter is parts and spent most of its time there. The bottleneck are computation parts and I/O parts. Because in sequential model, although no dependency between each other, all the urls need stand in a line, waiting all previous urls to finish. I think i parallized the hotspots by dividing and assigning computation of url blocks to different threads. These blocks is processed concurrently with litte shared data. Also, I think I removed bottlenecks. Firstly, as what I said above, the parallel model do computation concurrently. Secondly, I/O and computation can be done concurrently. Parsing is done immediately after the question channel has data, rather than after reading finished.

4.2 Challenges and Limits

The first challenge is decomposition. Because the main task of the problem lies in computation parts. Although the main thread can do I/O stuff while worker threads do computation, main thread can finish reading quickly then be idle for a long while. So how to reuse main thread or how to help worker threads to do computation is challenging. The second challenge is modeling. This task can also be finished really easily by reading all urls then evenly parsing urls. But this didn't involve functional decomposition and makes less use of computation resourceh. Assume the file is quite large, it would achieve a bad perfomance by spending too much time and space on I/O. So how to build the model to make best use of computation resource is also a challenge and aslo a important problem.
As for limits, I think i parallized the task pretty well. The speedup is pretty good. Of course, the hardware is a crucial problem to solve for more speedup. Also, communication overhead is also a important problem. So that's why i setup url blocks to adjust granularity. Choosing the better block size can lead to better performance. I draw a picture for different blocks(Figure 2), and it reflects the

relationship between granularity and speedup. Here, I used 100 urls and 4 threads. From the figure, we can see best block size is 20. Too large block size will cause too coarse-grained granularity and too small block size will cause too fined-grained granularity.

5 Advanced Features

In this project, I implement three advanced features. Firstly, I empplyed both functional and data decomposition components. For functional decomposition. I have the main thread to do reading and constructing the result map, also have worker threads to do url parsing. For data decomposition, I have every worker thread to do same calculations for different and independent parts of the input file. Secondly, I added a load balancing to the system. In the woker goroutines, I implemented the worker function by a for loop. So every thread can be reused and keep as busy as possible. Suppose one thread finished one block much faster than others, instead of be idle, it would be reused to process another block. So in this way, different threads might take different amount of work, but they can reach a barrier in a short time peroid because of dynamic load balancing. Lastly, I explicitly used error-handling pattern in parsing url , Here, 'questionChan' takes the role of 'result' and 'tagChan' takes the role of 'done' channel. When all thread finish processing urls, 'tagChan' will be closed. Also, I used the template of 'pi' covered in lecture 7 to handle main thread tasks in worker function.