SCRAWLER: A SEED-BY-SEED PARALLEL WEB CRAWLER

Joo Yong Lee, Sang Ho Lee
School of Computing, Soongsil University, Seoul, Korea

Yanggon Kim
Computer and Information Sciences, Towson University, Maryland, USA

Keywords: Web crawler, Parallel crawler, Scalability, Web database.

Abstract: As the size of the Web grows, it becomes increasingly important to parallelize a crawling process in order to complete downloading pages in a reasonable amount of time. This paper presents the design and implementation of an effective parallel web crawler. We first present various design choices and strategies for a parallel web crawler, and describe our crawler's architecture and implementation techniques. In particular, we investigate the URL distributor for URL balancing and the scalability of our crawler.

1 INTRODUCTION

A web crawler is a program that retrieves and stores web pages from the Web. A web crawler starts off by placing an initial set of URLs in a seed queue. The web crawler gets a URL from the seed queue, downloads the web page, extracts any URLs in the downloaded page, puts the new URLs in the seed queue, and gets the next URL from the seed queue. The web crawler repeats this crawling process until it decides to stop. Figure 1 illustrates the crawling process.

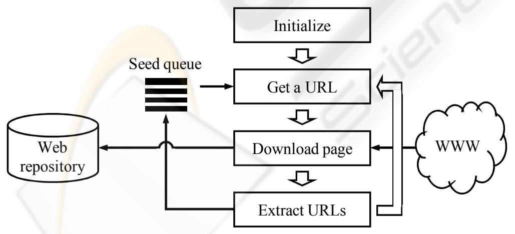

Figure 1: Flow of a crawling process.

A web crawler often has to download thousands of millions of pages in a short period of time and has to constantly monitor and refresh the downloaded pages. In addition, the web crawler should not put a severe burden on the visited web sites. As the size of the Web grows, it becomes more difficult or impossible to crawl the entire or significant portion of the Web by a single crawling process. In order to maximize the download rate, many web search engines run multiple crawling processes in parallel. We refer to this type of web crawler as a parallel web crawler.

Web crawlers have been studied since the advent of the Web. The first web crawler is Wanderer (Gray, M., 1996). The Internet Archive (Burner, M., 1997) crawler uses multiple machines to crawl the Web. Also, it uses a site-by-site basis crawl and a Bloom filter to find duplicate URLs. The original Google crawler (Brin, S., Page, L., 1998) is a distributed system that uses multiple machines for web crawling. Typically, three to four crawling machines are used to crawl the Web, so the entire system requires between four and eight machines. Mercator (Heydon, A., Najork, M., 1999) presents a number of functional components that crawler's basic algorithm requires: storing the list of URLs to download, resolving host names into IP addresses, downloading documents using the HTTP protocol, extracting links from HTML documents, and determining whether a URL has been encountered before. WebBase (Cho, J., Garcia-Molina, H., Haveliwala, T., Lam, W., Paepcke, A., Raghavan, S., Wesley, G., 2006) is an experimental web repository built at Stanford University. The WebBase project has implemented an incremental crawler (Cho, J., Garcia-Molina, H., 2000). The incremental crawler can keep crawled pages after the first crawl and revisit only changed pages (with high probability).

This paper expands on Kim and Lee's work (Kim, S.J., Lee, S.H., 2003) in terms of parallelization and scalability of a web crawler. Although the web crawling process is conceptually simple, designing a parallel web crawler is a complex endeavour. In this paper, we present the design and implementation of SCrawler (Soongsil Crawler). SCrawler has the following characteristics:

Full distribution. SCrawler can be distributed across multiple crawling machines for better performance. Crawling machines download web pages independently without communication between them.

Scalability. Due to the fully distributed architecture of SCrawler, its performance can be scaled by adding extra machine. In addition, data structures of SCrawler use a limited amount of main memory, regardless of the size of the Web. Therefore, SCrawler can manage to handle the rapidly growing Web.

Extensibility. SCrawler has been designed in a modular way. For a particular crawling environment, SCrawler can be reconfigured by plugging in appropriate modules without modifying its core components.

Portability. SCrawler is written entirely in Java to achieve platform independence, thus runs on any platforms for which there exists a Java virtual machine.

2 DESIGN OF A PARALLEL WEB CRAWLER

In order to maximize the download rate, a parallel web crawler runs multiple crawling processes simultaneously. Figure 2 illustrates multiple crawling processes.

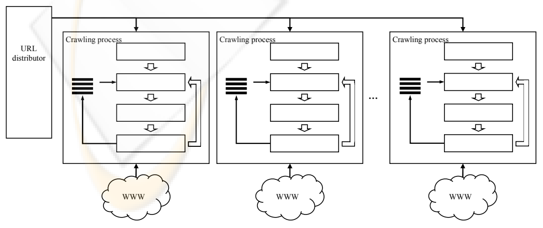

Figure 2: Flow of multiple crawling processes.

Generally, a web site is composed of more than one page. Given two web pages, *p* and *q*, that exist in the same web site, we say that *p* is locally reachable from *q* if *p* can be reachable from *q* through the web pages only in the same site. In this paper, a collection of web pages that are locally reachable from a given page (or seed page) is called a *party*.

2.1 Type of Distribution

A parallel web crawler can be "intra-site" or "distributed". An "intra-site" crawler runs all crawling processes on the same local network such as LAN. A "distributed" crawler runs crawling processes at geographically distant locations connected by the Internet or WAN. SCrawler is a "distributed" crawler that can run on any kind of network.

2.2 Coordination of Partition

When multiple crawling processes download web pages in parallel, different crawling processes may download the same page multiple times. A parallel web crawler can be classified into three ways in terms of handling this overlap. In the first scheme, "independent", crawling processes may download web pages independently without any coordination. Second, when there exists a central coordinator that logically divides the Web into small partitions and dynamically assigns each partition to a crawling process, we refer to this type of coordination as "dynamic assignment". Third, when the Web is partitioned and assigned to each crawling process before they start to crawl, we call it "static assignment".

SCrawler uses the "independent" scheme, thus has no coordination overhead and can be very scalable. In addition, since a seed-by-seed basis crawl downloads web pages in the same *party*, SCrawler can simulate the effect of "static assignment". Therefore, SCrawler can reduce more overlap than a parallel web crawler with "independent" scheme.

2.3 Crawling Mode

Each crawling process of a parallel web crawler is responsible for a pre-determined partition of the Web. Each crawling process of SCrawler has to download web pages in the party. A web crawler separates URLs into internal URLs and external URLs. Internal URLs represent web pages that belong to the partition, to which the downloaded page belongs. An external URL is a URL that is not an internal URL. There are three modes to handle external URLs.

In “firewall mode”, each crawling process downloads only web pages in its partition. All external URLs are ignored and thrown away, and thus all crawling processes may not download all web pages that it has to download. In “cross-over mode”, each crawling process can download web pages not only in its partition but also in external URLs. When crawling processes periodically and incrementally exchange external URLs, we call it as “exchange mode”.

SCrawler handles external URLs in “firewall mode” and overcomes the limitations of “firewall mode” by using a seed-by-seed basis crawl. We will discuss a seed-by-seed basis crawl in the next section.

3 IMPLEMENTATION ISSUES

SCrawler uses strategies such as a Bloom filter (Burner, M., 1997) to remove duplicate URLs and a breadth-first search to select web pages to be crawled. We decided to use Java(TM) 2 as implementation language to achieve platform independence. Also, we used MySQL to manage huge statistics from web crawling. SCrawler consists of 41 Java classes, with 292 methods and about 8,000 lines of code.

3.1 Seed-by-Seed Basis Crawl

A site-by-site basis crawl (Burner, M., 1997) is known to be a useful strategy. Since a web crawler with this approach maintains exactly one page (usually homepage) per a web site as a starting page of crawling, the web crawler cannot crawl web pages that are not connected from the homepage. In order to solve this problem, we have expanded the concept of a site-by-site basis crawl.

SCrawler crawls on a seed-by-seed basis (Kim, S.J., Lee, S.H., 2003). In a seed-by-seed basis crawl, it is possible for a single web site to have more than one seed. When there are web pages that are not locally reachable from a seed, SCrawler can crawl those pages from another seed. A single web site can have more than one party, and in this case SCrawler can maintain more than one seed for a single web site. Therefore, SCrawler finds web pages more than a web crawler with a site-by-site basis crawl does.

3.2 Crawling Process of SCrawler

Figure 3 shows the overall architecture of crawling process of SCrawler. Web crawling is performed by multiple crawling threads. Each crawling threads repeatedly performs the crawling process. Prior to a crawl, *Input URL Importer* stores seed URLs in *Input URL File* into a database. During the crawl, newly found URLs are stored in the database. After the crawl, *Output URL Exporter* writes URLs in the database into *Output URL File*. Therefore, *Output URL File* includes all URLs in *Input URL File*.

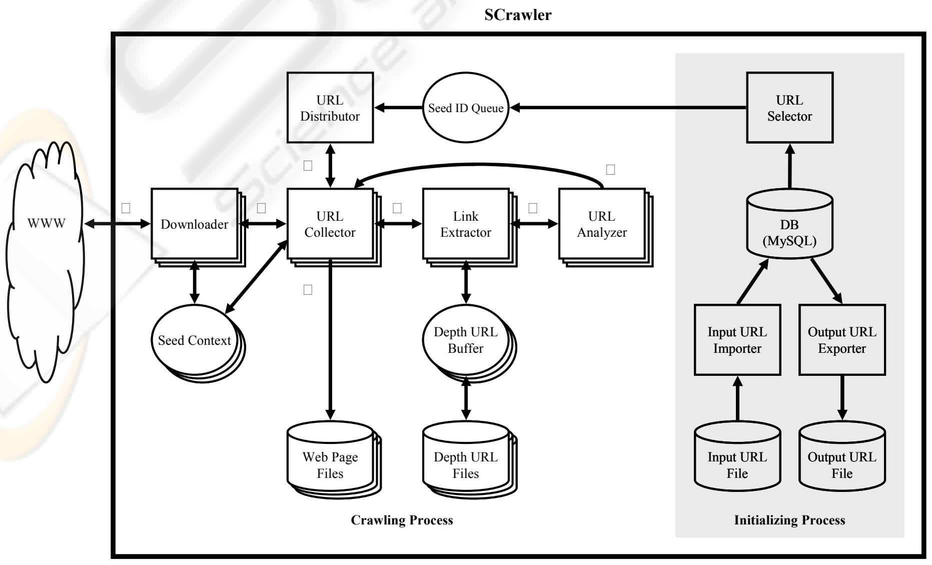

Figure 3: Main components of SCrawler.

SCrawler can use *Output URL File* as *Input URL File* on the next crawl. *URL Selector* selects seed URLs from the database and loads them into *Seed ID Queue*.

The first step of the crawling process is that *URL Collector* gets a seed URL from *Seed ID Queue* and determines whether it should crawl the seed URL (1). *URL Collector* makes *Seed Context* for a party of the seed URL. *Seed Context* is a data structure that contains a number of seed information such as an IP address, a seed URL string, a port number, a Bloom filter for the *party*, an elapsed time *"e-time"* after the last downloading and more. When *URL Collector* crawls multiple seed URLs simultaneously, it has multiple *Seed Context*s for each *party* of seed URLs.

After making *Seed Context*, *URL Collector* calls *Downloader* (2). *Downloader* downloads a web page using information of *Seed Context* (3). *URL Collector* stores the downloaded page in disk (4), and sends it to *Link Extractor* (5). *Link Extractor* extracts any URLs in the downloaded page. The extracted URLs are stored in *Depth URL Buffer* temporarily. If *Depth URL Buffer* is full, all URLs in the buffer are stored in *Depth URL File* and the buffer becomes clear.

URL Analyzer examines whether the extracted URLs are in syntax errors, and checks duplicate URLs (6). *URL Analyzer* separates the extracted URLs into internal URLs and external URLs. Internal URLs are sent to URL Collector (7). External URLs become candidate seed URLs. SCrawler repeats from step (2) to step (7) until no page URL in the *party* is found.

3.3 URL Balancing

SCrawler runs multiple crawling processes in parallel by multiple machines or multiple threads. Crawling threads download web pages independently without communication between them. Each crawling thread gets a seed URL from a seed queue. At any crawling times, the number of seed URLs that are assigned to each crawling thread should be maintained to be equal as much as possible (we call it *URL balancing)*. SCrawler uses *URL Distributor* to achieve *URL balancing*. Figure 4 conceptually shows the distribution of seed URLs by *URL Distributor*.

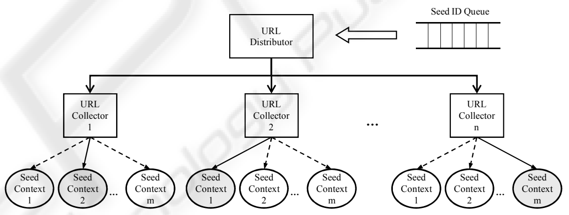

Figure 4: Conceptual diagram of distribution of seed URLs by *URL Distributor*.

URL distribution algorithm

```
while (true)
    if (not empty(seed_queue)) then
        foreach t in all_threads
            if (not full(seeds_of_t)) then
                url = dequeue(seed_queue)
                distribute(t, url)
                countSeed(t)
    else
        src_thread = findMaxSeed(all_threads)
        dst_thread = findMinSeed(all_threads)
        if (min_seed < avg_seed) then
            url = selectToDistribute(src_thread)
            distribute(dst_thread, url)
            countSeed(src_thread)
            countSeed(dst_thread)
```

Function description

`dequeue(queue)`: Remove a seed URL from `queue`, and return it.
`distribute(thread,url)`: Distribute `url` to `thread`, and make *seed context* for a *party* of `url`.
`countSeed(thread)`: Count seed URLs in `thread`.
`findMaxSeed(threads)`: Return the thread that has the most seed URLs in `threads`.
`findMinSeed(threads)`: Return the thread that has the least seed URLs in `threads`.
`selectToDistribute(thread)`: Select an inactive seed in `thread`, and return it.

Figure 5: URL distribution algorithm for *URL Distributor*.

URL Distributor uses the URL distribution algorithm that is illustrated in Figure 5. URL Distributor maintains information on the number of seed URLs that each crawling thread holds and the average number of seed URLs that all crawling threads hold. URL Distributor gets a seed URL from Seed ID Queue and distributes the seed URL to each crawling thread in a sequential order. Given a seed URL, the number of downloaded pages or the crawling time completely depends on the seed URL. Therefore, although all crawling threads start with the same number of seed URLs, their crawling time is likely to be different. For this reason, the number of seed URLs in each crawling thread is likely to be unbalanced after all seed URLs in Seed ID Queue have been evenly distributed. In order to cope with such URL unbalancing, URL Distributor redistributes seed URLs that crawling threads hold each other when some condition is met (see line 11).

We performed an experiment on the efficiency of *URL Distributor*. We considered the distribution of seed URLs in free competition as the object of comparison. The distribution of seed URLs in free competition means that each *URL Collector* gets seed URLs from *Seed ID Queue* in free competition. This distribution is illustrated in Figure 6.

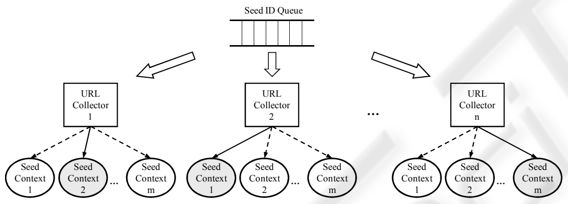

Figure 6: Conceptual diagram of distribution of seed URLs in free competition.

We randomly selected 10,000 Korean web sites and used them as seeds. SCrawler ran 20 crawling threads simultaneously, each of which held 10 *Seed Contexts*. We conducted web crawling twice. The first crawl distributed seed URLs by *URL Distributor*. In the second crawl, seed URLs were distributed in free competition.

Figure 7 and 8 show the number of seed URLs that each of 20 crawling threads holds. In the figures, bold lines represent the average number of seed URLs that all crawling threads hold. From Figure 7, we can see that after 40 seconds of crawling, the average number of seed URLs all crawling threads held was maintained to be approximately 9. From Figure 8, we can observe that thread 18 (T18) held 10 seed URLs after 100 seconds of crawling and thread 0 (T0) held only one seed URL until 60 seconds of crawling. Seed URLs were not equally distributed to each thread in free competition.

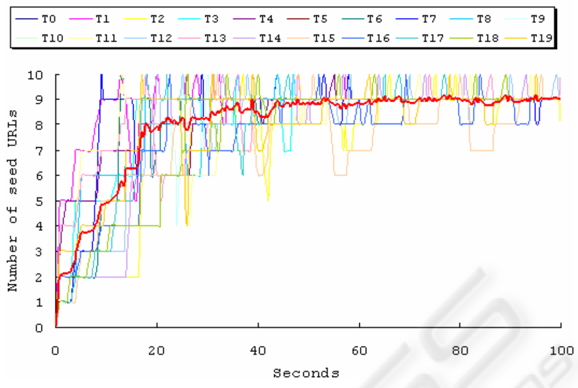

Figure 7: Number of seed URLs in *URL Distributor*.

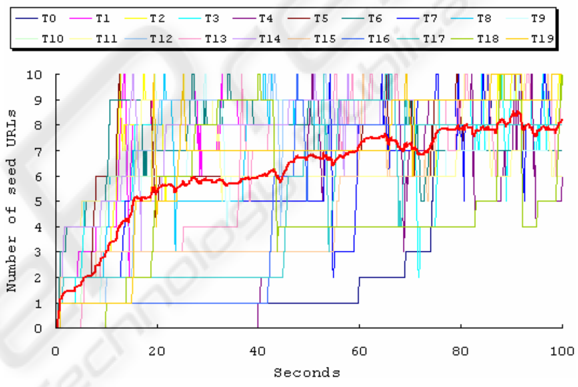

Figure 8: Number of seed URLs in free competition.

Using URL Distributor, we reduced crawling time as much as 2 hours and 40 minutes than that in free competition. The download rate also increased by 34.7%. The results show that the URL distribution algorithm works well in multiple crawling processes.

3.4 Scalability

In a highly scalable system, one should guarantee that the work performed by every thread is constant as the number of threads changes (Boldi, P., Codenotti, B., Santini, M., Vigna, S., 2004). That is, the system and communication overheads do not reduce the performance of each thread. SCrawler is scalable in two respects. First, SCrawler has a fully distributed architecture. Due to this architecture, the performance of SCrawler is scaled by adding extra machines. Second, data structures of SCrawler use a bounded amount of main memory, regardless of the size of the Web. Therefore, SCrawler can cope with the rapidly growing Web.

We experimentally measured how the average number of downloaded pages per second per thread changes as the number of crawling machines changes. For this, we used 250,000 Korean sites randomly selected as seeds. We increased crawling machines from one to five by increment of one. We set the number of crawling threads for each machine to 10, 15 and 20, and each thread ran with 10 seeds simultaneously.

Figure 9 shows how the average number of pages downloaded per second per thread changes as the number of crawling machines increases. The solid line, long dashes, and short dashes represent web crawling using 20 threads, 15 threads, and 10 threads, respectively. The more systems are scalable, the more lines are horizontal. From the results, we believe that SCrawler is scalable almost linearly with the number of crawling machines. One might notice that the lines are not completely horizontal. This could be attributed to the limitations of our network resources. We ran this experiment in a campus network where the network status is likely to vary over time.

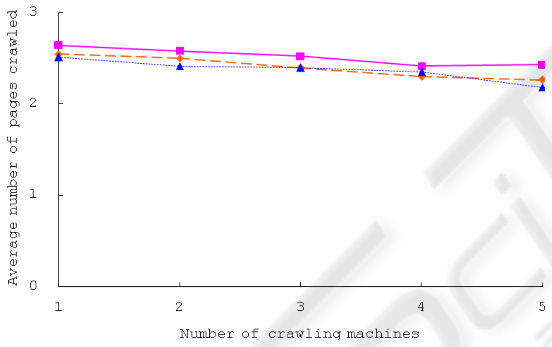

Figure 9: Average number of pages crawled per second per thread.

4 CLOSING REMARKS

The development of SCrawler is ongoing. Dynamically generated contents are constantly created on the Web. The Web is growing tremendously. Our next expansion of SCrawler would be to selectively crawl web pages that are relevant to a pre-defined set of topics.

ACKNOWLEDGEMENTS

This work was supported by Seoul R&BD Program (10581cooperateOrg93112).

REFERENCES

Boldi, P., Codenotti, B., Santini, M., Vigna, S., 2004. UbiCrawler: a scalable fully distributed Web crawler. *Software-Practice and Experience*, Vol. 34, No. 8, 711-726.

Brin, S., Page, L., 1998. The anatomy of a large-scale hypertextual Web search engine. In *Computer Networks and ISDN Systems*, Vol. 30, No.1-7, 107-117.

Burner, M., 1997. Crawling towards Eternity: Building An Archive of The World Wide Web. In *Web Techniques Magazine*, Vol. 2, No. 5, 37-40.

Cho, J., Garcia-Molina, H., 2002. Parallel Crawlers. In *WWW'02, 11th International World Wide Web Conference*, 124-135.

Cho, J., Garcia-Molina, H., 2000. The Evolution of the Web and Implications for an Incremental Crawler. In *VLDB'00, 26th International Conference on Very Large Data Bases*, 200-209.

Cho, J., Garcia-Molina, H., Haveliwala, T., Lam, W., Paepcke, A., Raghavan, S., Wesley, G., 2006. Stanford WebBase Components and Applications. In *ACM Transactions on Internet Technology*. Vol. 6, No. 2, 153-186.

Gray, M., 1996. *Internet Statistics: Growth and Usage of the Web and the Internet*, http://www.mit.edu/people/mkgray/net/.

Heydon, A., Najork, M., 1999. Mercator: A scalable, extensible Web crawler. In *World Wide Web*, Vol. 2, No. 4, 219-229.

Kim, S.J., Lee, S.H., 2003. Implementation of a Web Robot and Statistics on the Korean Web. In *HSI'03, 2nd International Conference of Human.Society@ Internet*, 341-350.

Najork, M., Heydon, A., 2001. High-performance web crawling. In *SRC Research Report 173*. Compaq Systems Research Center.

Najork, M., Wiener, J.L., 2001. Breadth-First Search Crawling Yields High-Quality Pages. In *WWW'01, 10th International World Wide Web Conference*, 114-118.

Shkapenyuk, V., Suel, T., 2002. Design and Implementation of a High-Performance Distributed Web Crawler. In *ICDE'02, 18th International Conference on Data Engineering*, 357-368.