Crawling the web: web crawling and social media crawling

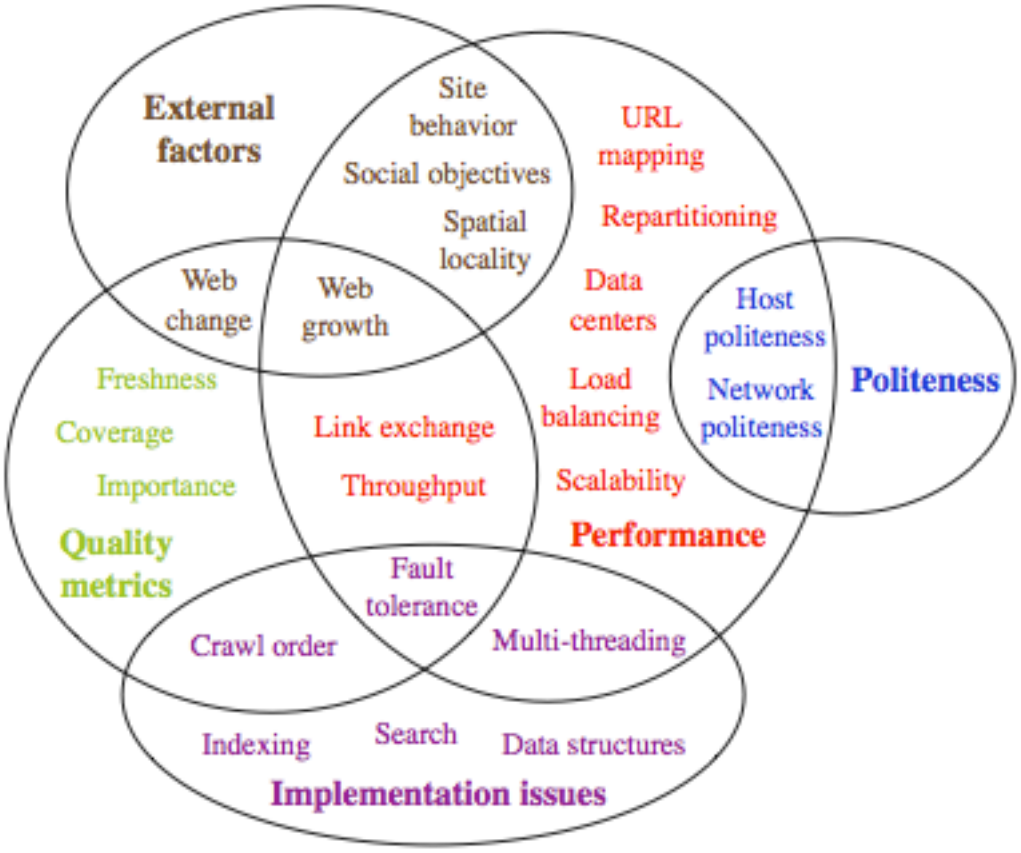
Background:

I am a fourth-year undergraduate EECS major at UC Berkeley. Overall the focus of my studies has been around building scalable social applications and all the steps that requires. I am interested in the ways in which web applications can be used to create social change or assist people's everyday living. Seeing as parallelism is a key component in design of making scalable architecture, I wanted to take this course to learn more about aspects of parallelism that would apply, such as in parallel search engines, database optimizations, and the cloud. And also, that regardless of the job I took afterwards in the computer science umbrella, to learn when applying parallelism would be useful, vs when it is too much overhead, would be essential to understand in a workplace environment. I am good at coding in C, prefer coding in python/ruby, and have taken databases, algorithms, artificial intelligence, embedded systems, and advanced signal processing courses.

Web crawler basics:

Web crawlers are essential for building search engines that can traverse across the web and provide intelligible content to users. Typically a crawler's purpose is to create a copy of all visited pages for further processing and easy access later on, as well as used to process said documents of information valuable to the intent of crawling the web (be that searching for email addresses, grabbing technical information, etc). Given the billions upon billions of web pages on the web, with some crawlers claiming to be going through tens of billions of web pages a day.

Despite seeming like a simple task at first (traversing through web pages may sound like trivial graph traversal), it comes with a series of hitches even as a sequential problem:



Priority issues: what kind of information and links are considered valuable? While there are some clever algorithms out there that attempt to address issues of popularity (such as PageRank), there are special considerations to consider if say, you are looking for a clustered set of information on a certain topic as opposed to just mining the entire web.

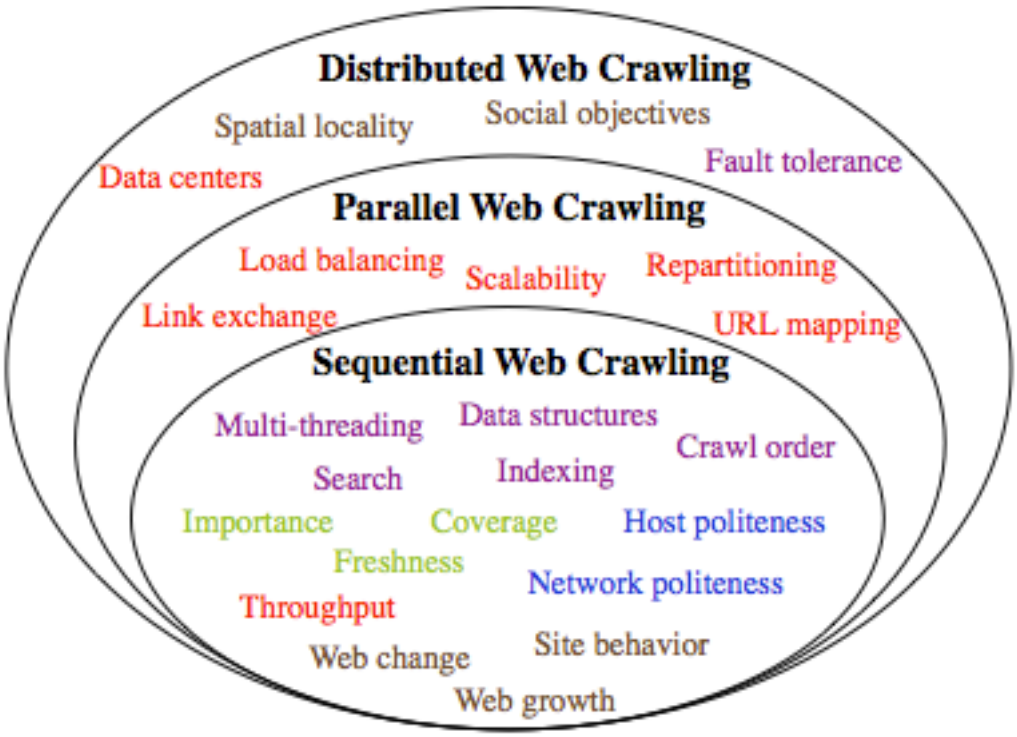
Revisiting policy: given the fact that the web is always dynamically changing, how can we ensure not to penalize new content over old content, and how to deal with the fact new content may have developed on a page whilst in the middle of a search?

Politeness policy: the amount of resources behind a powerful web crawler could very well down servers not expecting a high load of traffic as a crawler can search through and download pages far faster than an ordinary web-surfing user. How can does one guarantee availability of the page for others?

Storage: where does one store the information downloaded? How is it organized in a fashion that can be used to search through later?

Challenges of building a good parallel crawler:

In addition to the above mentioned issues, a parallel crawler can take of one of two forms: a distributed crawler, or a parallel cluster. Both have a very interesting set of challenges:



Load balancing and network exchanges of links are a very difficult balance for parallelization of web crawling. A general division of work for processors is to have a series of threads crawling the frontier of the web on the behest of supervising computers, which, after downloading and processing any new URLs, pass that information back up to the master server to send new demands. This requires a lot of overhead with out-of-core processing and message passing, as while one attempts to limit the amount of communication between servers around the URLs processed, due to the connected nature of the web it is necessary to establish a common policy of communication between threads.

Crawling politely also becomes much more difficult to manage with distributed and parallel computing, especially with distributed computing. A typical tactic for good load balance is distribution of domains to different workers, but that does have the tradeoff of potentially losing some links in the process. Order of crawling is also very significant, as it determines which pages will get seen first and prioritized over those that do not. And once the data is collected, sharing the data across servers becomes a very important thing to manage.

In order to build a parallel web crawler, one needs a variety of coding platforms in place in order to make an effective crawler:

1: a system for managing workers and setting tasks between processors, often through higher-level approaches such as Redis or amazon's EC2 web servers
2: a database language and method for storing objects into the database, such as SQL or mongoDB. Typical data structures used are hash sets, sorted sets, and Bloom filters (with tradeoffs between speed and correctness, often having to move into more abstract data sets such as Bloom filters based off the scope of how many web pages one is scraping)
3: Scripts and algorithms for easily searching through the collected information to help prioritize web pages and grab relevant information. One example of such a platform for SQL would be solr.

Simple implementations using clusters of 20 to 60 computers have been proven to crawl pages at rates of 1-3 million a day to a quarter billion pages in 40 hours with 60 pages. Such examples show the potential for scalability with parallel web crawlers, despite the challenges faced by developing such an interface.

Within the systems that I evaluated, the main point of weakness and bottleneck was the messaging and communication aspect of parallel processing—in one of the examples I studied, there was a severe bottleneck at the fact that there was only one supervisor for all the workers. There also are bottlenecks and issues around network connectivity, downloading, and parsing the web pages. Particularly once one starts to scale to even more computers and processors, the importance of sharing domains cleverly and addressing the potential for failure of network requests becomes more relevant.

I am particularly interested in the ways in which said web crawls, when looking for say, academic papers as opposed to a general web search, become a problem that really begs the need of parallel processing in order to scrape a good portion of the ever-growing internet in a fast matter. I am also interested by the problems one gets not just from merely parallelizing it, but by attempting to allow users to volunteer time into the search engine to create a distributed search engine. It also has interesting applications within social apps, where one could be say, searching for information about individuals or organizations as opposed to web links or pdf files.

Cambazoglu, Barla, Flavio Junqueira, Vassilis Plachouras, and Luca Telloli. "On the Feasibility of Geographically Distributed Web Crawling." Yahoo, 2008. Web. 5 Feb. 2013. <http://research.yahoo.com/pub/2228>.

Duen, Chau, Shashank Pandit, Samuel Wang, and Christos Faloutsos. "Parallel Crawling for Online Social Networks ." Carnegie Melon University, 2007. Web. 6 Feb. 2013. <http://www.www2007.org/posters/poster1057.pdf>.

Nielsen, Michael. "How to crawl a quarter billion webpages in 40 hours." DDI, 10 Aug. 2012. Web. 7 Feb. 2013. <http://www.michaelnielsen.org/ddi/how-to-crawl-a-quarter-billion-webpages-in-40-hours/>.

Seeger, Marc. "Building blocks of a scalable web crawler." Stuttgart Media University, 10 Sept. 2010. Web. 5 Feb. 2013. <http://blog.marc-seeger.de/assets/papers/thesis_seeger-building_blocks_of_a_scalable_webcrawler.pdf>.

"Threaded/Parallel Web Crawler (or Web Server Killing Software)." yPass, 26 Jan. 2010. Web. 4 Feb. 2013. <http://www.ypass.net/blog/2010/01/threadedparallel-web-crawler-or-web-server-killing-software/>.

Varun, Sivamani. "How we built our 60-node (almost) Distributed Web Crawler." N.p., 3 Sept. 2012. Web. 6 Feb. 2013. <http://blog.semantics3.com/how-we-built-our-almost-distributed-web-crawler/>.