Parallel Crawlers

Junghoo Cho
University of California, Los Angeles
cho@cs.ucla.edu

Hector Garcia-Molina
Stanford University
cho@cs.stanford.edu

ABSTRACT

In this paper we study how we can design an effective parallel crawler. As the size of the Web grows, it becomes imperative to parallelize a crawling process, in order to finish downloading pages in a reasonable amount of time. We first propose multiple architectures for a parallel crawler and identify fundamental issues related to parallel crawling. Based on this understanding, we then propose metrics to evaluate a parallel crawler, and compare the proposed architectures using 40 million pages collected from the Web. Our results clarify the relative merits of each architecture and provide a good guideline on when to adopt which architecture.

Keywords

Web Crawler, Web Spider, Parallelization

1. INTRODUCTION

A crawler is a program that downloads and stores Web pages, often for a Web search engine. Roughly, a crawler starts off by placing an initial set of URLs, S_0, in a queue, where all URLs to be retrieved are kept and prioritized. From this queue, the crawler gets a URL (in some order), downloads the page, extracts any URLs in the downloaded page, and puts the new URLs in the queue. This process is repeated until the crawler decides to stop. Collected pages are later used for other applications, such as a Web search engine or a Web cache.

As the size of the Web grows, it becomes more difficult to retrieve the whole or a significant portion of the Web using a single process. Therefore, many search engines often run multiple processes in parallel to perform the above task, so that download rate is maximized. We refer to this type of crawler as a *parallel crawler*.

In this paper we study how we should design a parallel crawler, so that we can maximize its performance (e.g., download rate) while minimizing the overhead from parallelization. We believe many existing search engines already use some sort of parallelization, but there has been little scientific research conducted on this topic. Thus, little has been known on the tradeoffs among various design choices for a parallel crawler. In particular, we believe the following issues make the study of a parallel crawler challenging and interesting:

WWW2002, May 7–11, 2002, Honolulu, Hawaii, USA.
ACM 1-58113-449-5/02/0005.

- *Overlap:* When multiple processes run in parallel to download pages, it is possible that different processes download the same page multiple times. One process may not be aware that another process has already downloaded the page. Clearly, such multiple downloads should be minimized to save network bandwidth and increase the crawler's effectiveness. Then how can we coordinate the processes to prevent overlap?
- *Quality:* Often, a crawler wants to download "important" pages first, in order to maximize the "quality" of the downloaded collection. However, in a parallel crawler, each process may not be aware of the whole image of the Web that they have collectively downloaded so far. For this reason, each process may make a crawling decision solely based on *its own* image of the Web (that itself has downloaded) and thus make a poor crawling decision. Then how can we make sure that the *quality* of the downloaded pages is as good for a parallel crawler as for a centralized one?
- *Communication bandwidth:* In order to prevent overlap, or to improve the quality of the downloaded pages, crawling processes need to periodically communicate to coordinate with each other. However, this communication may grow significantly as the number of crawling processes increases. Exactly what do they need to communicate and how significant would this overhead be? Can we minimize this communication overhead while maintaining the effectiveness of the crawler?

While challenging to implement, we believe that a parallel crawler has many important advantages, compared to a single-process crawler:

- *Scalability:* Due to enormous size of the Web, it is often imperative to run a parallel crawler. A single-process crawler simply cannot achieve the required download rate in certain cases.
- *Network-load dispersion:* Multiple crawling processes of a parallel crawler may run at geographically distant locations, each downloading "geographically-adjacent" pages. For example, a process in Germany may download all European pages, while another one in Japan crawls all Asian pages. In this way, we can *disperse* the network load to multiple regions. In particular, this dispersion might be necessary when a single network cannot handle the heavy load from a large-scale crawl.

- *Network-load reduction:* In addition to the dispersing load, a parallel crawler may actually *reduce* the network load. For example, assume that a crawler in North America retrieves a page from Europe. To be downloaded by the crawler, the page first has to go through the network in Europe, then the Europe-to-North America inter-continental network and finally the network in North America. Instead, if a crawling process in Europe collects all European pages, and if another process in North America crawls all North American pages, the overall network load will be reduced, because pages go through only "local" networks.

  Note that the downloaded pages may need to be transferred later to a central location, so that a central index can be built. However, even in that case, we believe that the transfer can be significantly smaller than the original page download traffic, by using some of the following methods:

  - *Compression:* Once the pages are collected and stored, it is easy to *compress* the data before sending them to a central location.
  - *Difference:* Instead of sending the entire image with all downloaded pages, we may first take *difference* between previous image and the current one and send only this difference. Since many pages are static and do not change very often, this scheme can significantly reduce the network traffic.
  - *Summarization:* In certain cases, we may need only a central index, not the original pages themselves. In this case, we may extract the necessary information for the index construction (e.g., postings list) and transfer this data only.

To build an effective web crawler, we clearly need to address many more challenges than just parallelization. For example, a crawler needs to figure out how often a page changes and how often it would revisit the page in order to maintain the page up to date [7, 9]. Also, it has to make sure that a particular Web site is not flooded with its HTTP requests during a crawl [16, 11, 26]. In addition, it has to carefully select what page to download and store in its limited storage space in order to make the best use of its stored collection of pages [8, 5, 10]. While all of these issues are important, we focus on the crawler parallelization in this paper, because this problem has been paid significantly less attention than the others.

In summary, we believe a parallel crawler has many advantages and poses interesting challenges. In particular, we believe our paper makes the following contributions:

- We identify major issues and problems related to a parallel crawler and discuss how we can solve these problems.
- We present multiple techniques for a parallel crawler and discuss their advantages and disadvantages. As far as we know most of these techniques have not been described in open literature. (Very little is known about the internals of crawlers, as they are closely guarded secrets.)
- Using a large dataset (40M web pages) collected from the Web, we experimentally compare the design choices and study their tradeoffs quantitatively.
- We propose various optimization techniques that can minimize the coordination effort between crawling processes, so that they can operate more independently while maximizing their effectiveness.

1.1 Related work

Web crawlers have been studied since the advent of the Web [18, 24, 4, 23, 13, 6, 19, 11, 8, 5, 10, 9, 7]. These studies can be roughly categorized into one of the following topics:

- *General architecture* [23, 13, 6, 19, 11]: The work in this category describes the general architecture of a Web crawler and studies how a crawler works. For example, Reference [13] describes the architecture of the Compaq SRC crawler and its major design goals. Some of these studies briefly describe how the crawling task is parallelized. For instance, Reference [23] describes a crawler that distributes individual URLs to multiple machines, which download Web pages in parallel. The downloaded pages are then sent to a central machine, on which links are extracted and sent back to the crawling machines. However, these studies do not carefully compare various issues related to a parallel crawler and how design choices affect performance. In this paper, we first identify multiple techniques for a parallel crawler and compare their relative merits using real Web data.
- *Page selection* [8, 5, 10]: Since many crawlers can download only a small subset of the Web, crawlers need to carefully decide what page to download. By retrieving "important" or "relevant" pages early, a crawler may improve the "quality" of the downloaded pages. The studies in this category explore how a crawler can discover and identify "important" pages early, and propose various algorithms to achieve this goal. In our paper, we study how parallelization affects some of these techniques and explain how we can fix the problems introduced by parallelization.
- *Page update* [9, 7]: Web crawlers need to update the downloaded pages periodically, in order to maintain the pages up to date. The studies in this category discuss various *page revisit policies* to maximize the "freshness" of the downloaded pages. For example, Reference [7] studies how a crawler should adjust *revisit frequencies* for pages when the pages change at different rates. We believe these studies are orthogonal to what we discuss in this paper.

There also exists a significant body of literature studying the general problem of parallel and distributed computing [20, 22, 25, 28]. Some of these studies focus on the design of efficient parallel algorithms. For example, References [25, 21, 15] present various architectures for parallel computing, propose algorithms that solve various problems (e.g., finding maximum cliques) under the architecture, and study the complexity of the proposed algorithms. While the general principles described are being used in our work,[1] none of the existing solutions can be directly applied to the crawling problem.

[1]For example, we may consider that our proposed solution is a variation of "divide and conquer" approach, since we partition and assign the Web to multiple processes.

Another body of literature designs and implements *distributed operating systems*, where a process can use distributed resources transparently (e.g., distributed memory, distributed file systems) [28, 27, 1, 17]. Clearly, such OS-level support makes it easy to build a general distributed application, but we believe that we cannot simply run a centralized crawler on a distributed OS to achieve parallelism. A web crawler contacts millions of web sites in a short period of time and consumes extremely large network, storage and memory resources. Since these loads push the limit of existing hardwares, the task should be carefully partitioned among processes and they should be carefully coordinated. Therefore, a general-purpose distributed operating system that does not understand the semantics of web crawling will lead to unacceptably poor performance.

2. ARCHITECTURE OF A PARALLEL CRAWLER

In Figure 1 we illustrate the general architecture of a parallel crawler. A parallel crawler consists of multiple crawling processes, which we refer to as C-proc's. Each C-proc performs the basic tasks that a single-process crawler conducts. It downloads pages from the Web, stores the pages locally, extracts URLs from the downloaded pages and follows links. Depending on how the C-proc's split the download task, some of the extracted links may be sent to other C-proc's. The C-proc's performing these tasks may be distributed either on the same local network or at geographically distant locations.

- *Intra-site parallel crawler:* When all C-proc's run on the same local network and communicate through a high speed interconnect (such as LAN), we call it an *intra-site parallel crawler*. In Figure 1, this scenario corresponds to the case where all C-proc's run only on the local network on the top. In this case, all C-proc's use the same local network when they download pages from remote Web sites. Therefore, the network load from C-proc's is centralized at a single location where they operate.
- *Distributed crawler:* When C-proc's run at geographically distant locations connected by the Internet (or a wide area network), we call it a *distributed crawler*. For example, one C-proc may run in the US, crawling all US pages, and another C-proc may run in France, crawling all European pages. As we discussed in the introduction, a distributed crawler can *disperse* and even *reduce* the load on the overall network.

  When C-proc's run at distant locations and communicate through the Internet, it becomes important how often and how much C-proc's need to communicate. The bandwidth between C-proc's may be limited and sometimes unavailable, as is often the case with the Internet.

When multiple C-proc's download pages in parallel, different C-proc's may download the same page multiple times. In order to avoid this overlap, C-proc's need to coordinate with each other on what pages to download. This coordination can be done in one of the following ways:

- *Independent:* At one extreme, C-proc's may download pages totally independently without any coordination.

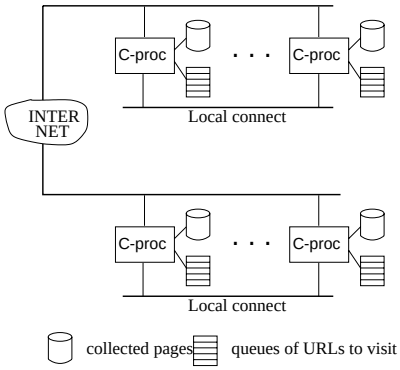


Figure 1: General architecture of a parallel crawler

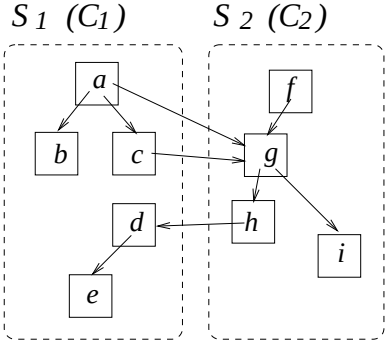


Figure 2: Site S_1 is crawled by C_1 and site S_2 is crawled by C_2

That is, each C-proc starts with its own set of seed URLs and follows links without consulting with other C-proc's. In this scenario, downloaded pages may overlap, but we may hope that this overlap will not be significant, if all C-proc's start from different seed URLs.

While this scheme has minimal coordination overhead and can be very scalable, we do not directly study this option due to its overlap problem. Later we will consider an improved version of this option, which significantly reduces overlap.

- *Dynamic assignment:* When there exists a central coordinator that logically divides the Web into small partitions (using a certain partitioning function) and dynamically assigns each partition to a C-proc for download, we call it *dynamic assignment*.

  For example, assume that a central coordinator partitions the Web by the site name of a URL. That is, pages in the same site (e.g., `http://cnn.com/top.html` and `http://cnn.com/content.html`) belong to the same partition, while pages in different sites belong to different partitions. Then during a crawl, the central coordinator constantly decides on what partition to crawl next (e.g., the site `cnn.com`) and sends URLs within this partition (that have been discovered so far) to a C-proc as seed URLs. Given this request, the C-proc downloads the pages and extracts links from them. When the extracted links point to pages in the same partition (e.g., `http://cnn.com/article.html`), the C-proc follows the links, but if a link points to a page in another partition (e.g., `http://nytimes.com/`

index.html), the C-proc reports the link to the central coordinator. The central coordinator later uses this link as a seed URL for the appropriate partition.

Note that the Web can be partitioned at various granularities. At one extreme, the central coordinator may consider every page as a separate partition and assign individual URLs to C-proc's for download. In this case, a C-proc does not follow links, because different pages belong to separate partitions. It simply reports all extracted URLs back to the coordinator. Therefore, the communication between a C-proc and the central coordinator may vary dramatically, depending on the granularity of the partitioning function.

- *Static assignment:* When the Web is partitioned and assigned to each C-proc *before* they start to crawl, we call it *static assignment*. In this case, every C-proc knows which C-proc is responsible for which page during a crawl, and the crawler does not need a central coordinator. We will shortly discuss in more detail how C-proc's operates under this scheme.

In this paper, we mainly focus on static assignment because of its simplicity and scalability, and defer the study of dynamic assignment to future work. Note that in dynamic assignment, the central coordinator may become the major bottleneck, because it has to maintain a large number of URLs reported from all C-proc's and has to constantly coordinate all C-proc's. Thus the coordinator itself may also need to be parallelized.

3. CRAWLING MODES FOR STATIC ASSIGNMENT

Under static assignment, each C-proc is responsible for a certain partition of the Web and has to download pages within the partition. However, some pages in the partition may have links to pages in another partition. We refer to this type of link as an *inter-partition link*. To illustrate how a C-proc may handle inter-partition links, we use Figure 2 as our example. In the figure, we assume two C-proc's, C_1 and C_2, are responsible for sites S_1 and S_2, respectively. For now, we assume that the Web is partitioned by sites and that the Web has only S_1 and S_2. Also, we assume that each C-proc starts its crawl from the root page of each site, a and f.

1. **Firewall mode:** In this mode, each C-proc downloads only the pages within its partition and does not follow any inter-partition link. All inter-partition links are ignored and thrown away. For example, the links $a \rightarrow g$, $c \rightarrow g$ and $h \rightarrow d$ in Figure 2 are ignored and thrown away by C_1 and C_2.

   In this mode, the overall crawler does not have any overlap in the downloaded pages, because a page can be downloaded by only one C-proc, if ever. However, the overall crawler may not download all pages that it has to download, because some pages may be reachable only through inter-partition links. For example, in Figure 2, C_1 can download a, b and c, but not d and e, because they can be reached only through $h \rightarrow d$ link. However, C-proc's can run quite independently in this mode, because they do not conduct any runtime coordination or URL exchanges.

2. **Cross-over mode:** Primarily, each C-proc downloads pages within its partition, but when it runs out of pages in its partition, it also follows inter-partition links. For example, consider C_1 in Figure 2. Process C_1 first downloads pages a, b and c by following links from a. At this point, C_1 runs out of pages in S_1, so it follows a link to g and starts exploring S_2. After downloading g and h, it discovers a link to d in S_1, so it comes back to S_1 and downloads pages d and e.

   In this mode, downloaded pages may clearly overlap (pages g and h are downloaded twice), but the overall crawler can download more pages than the firewall mode (C_1 downloads d and e in this mode). Also, as in the firewall mode, C-proc's do not need to communicate with each other, because they follow only the links discovered by themselves.

3. **Exchange mode:** When C-proc's periodically and incrementally exchange inter-partition URLs, we say that they operate in an *exchange mode*. Processes do not follow inter-partition links.

   For example, C_1 in Figure 2 informs C_2 of page g after it downloads page a (and c) and C_2 transfers the URL of page d to C_1 after it downloads page h. Note that C_1 does not follow links to page g. It only transfers the links to C_2, so that C_2 can download the page. In this way, the overall crawler can avoid overlap, while maximizing coverage.

Note that the firewall and the cross-over modes give C-proc's much independence (C-proc's do not need to communicate with each other), but they may download the same page multiple times, or may not download some pages. In contrast, the exchange mode avoids these problems but requires constant URL exchange between C-proc's.

3.1 URL exchange minimization

To reduce URL exchange, a crawler based on the exchange mode may use some of the following techniques:

1. **Batch communication:** Instead of transferring an inter-partition URL immediately after it is discovered, a C-proc may wait for a while, to collect *a set of URLs* and send them *in a batch*. That is, with batching, a C-proc collects all inter-partition URLs until it downloads k pages. Then it partitions the collected URLs and sends them to an appropriate C-proc. Once these URLs are transferred, the C-proc then purges them and starts to collect a new set of URLs from the next downloaded pages. Note that a C-proc does *not* maintain the list of *all* inter-partition URLs discovered so far. It only maintains the list of inter-partition links in the *current* batch, in order to minimize the memory overhead for URL storage.

   This *batch communication* has various advantages over incremental communication. First, it incurs less communication overhead, because a set of URLs can be sent in a batch, instead of sending one URL per message. Second, the absolute number of exchanged URLs will also decrease. For example, consider C_1 in Figure 2. The link to page g appears twice, in page a and in page c. Therefore, if C_1 transfers the link to g after downloading page a, it needs to send the same URL

again after downloading page c.[2] In contrast, if C_1 waits until page c and sends URLs in batch, it needs to send the URL for g only once.

2. **Replication:** It is known that the number of *incoming* links to pages on the Web follows a Zipfian distribution [3, 2, 29]. That is, a small number of Web pages have an extremely large number of links pointing to them, while a majority of pages have only a small number of incoming links.

   Thus, we may significantly reduce URL exchanges, if we replicate the most "popular" URLs at each C-proc (by most popular, we mean the URLs with most incoming links) and stop transferring them between C-proc's. That is, before we start crawling pages, we identify the most popular k URLs based on the image of the Web collected in a *previous* crawl. Then we replicate these URLs at each C-proc, so that the C-proc's do not exchange them during a crawl. Since a small number of Web pages have a large number of incoming links, this scheme may significantly reduce URL exchanges between C-proc's, even if we replicate a small number of URLs.

   Note that some of the replicated URLs may be used as the seed URLs for a C-proc. That is, if some URLs in the replicated set belong to the same partition that a C-proc is responsible for, the C-proc may use those URLs as its seeds rather than starting from other pages.

   Also note that it is possible that each C-proc tries to discover popular URLs *on the fly* during a crawl, instead of identifying them based on the previous image. For example, each C-proc may keep a "cache" of recently seen URL entries. This cache may pick up "popular" URLs automatically, because the popular URLs show up repeatedly. However, we believe that the popular URLs from a previous crawl will be a good approximation for the popular URLs in the current Web; Most popular Web pages (such as Yahoo) maintain their popularity for a relatively long period of time, even if their exact popularity may change slightly.

3.2 Partitioning function

So far, we have mainly assumed that the Web pages are partitioned by Web sites. Clearly, there exist multitude of ways to partition the Web, including the following:

1. **URL-hash based:** Based on the hash value of the URL of a page, we assign the page to a C-proc. In this scheme, pages in the same site can be assigned to different C-proc's. Therefore, the locality of link structure[3] is not reflected in the partition, and there will be many inter-partition links.

2. **Site-hash based:** Instead of computing the hash value on an entire URL, we compute the hash value only on the *site name* of a URL (e.g., `cnn.com` in `http://cnn.com/index.html`) and assign the page to a C-proc.

   In this scheme, note that the pages in the same site will be allocated to the *same* partition. Therefore, only some of the *inter-site* links will be *inter-partition* links, and thus we can reduce the number of inter-partition links quite significantly compared to the URL-hash based scheme.

3. **Hierarchical:** Instead of using a hash-value, we may partition the Web hierarchically based on the URLs of pages. For example, we may divide the Web into three partitions (the pages in the `.com` domain, `.net` domain and all other pages) and allocate them to three C-proc's. Even further, we may decompose the Web by language or country (e.g., `.mx` for Mexico).

   Because pages hosted in the same domain or country may be more likely to link to pages in the same domain, scheme may have even fewer inter-partition links than the site-hash based scheme.

In this paper, we do not consider the URL-hash based scheme, because it generates a large number of inter-partition links. When the crawler uses URL-hash based scheme, C-proc's need to exchange much larger number of URLs (exchange mode), and the coverage of the overall crawler can be much lower (firewall mode).

In addition, in our later experiments, we will mainly use the site-hash based scheme as our partitioning function. We chose this option because it is much simpler to implement, and because it captures the core issues that we want to study. For example, under the hierarchical scheme, it is not easy to divide the Web into equal size partitions, while it is relatively straightforward under the site-hash based scheme.[4] Also, we believe we can interpret the results from the site-hash based scheme as the upper/lower bound for the hierarchical scheme. For instance, assuming Web pages link to more pages in the same domain, the number of inter-partition links will be lower in the hierarchical scheme than in the site-hash based scheme (although we could not confirm this trend in our experiments).

In Figure 3, we summarize the options that we have discussed so far. The right-hand table in the figure shows more detailed view on the static coordination scheme. In the diagram, we highlight the main focus of our paper with dark grey. That is, we mainly study the static coordination scheme (the third column in the left-hand table) and we use the site-hash based partitioning for our experiments (the second row in the second table). However, during our discussion, we will also briefly explore the implications of other options. For instance, the firewall mode is an "improved" version of the independent coordination scheme (in the first table), so our study on the firewall mode will show the implications of the independent coordination scheme. Also, we roughly estimate the performance of the URL-hash based scheme (first row in the second table) when we discuss the results from the site-hash based scheme.

Given our table of crawler design space, it would be very interesting to see what options existing search engines selected for their own crawlers. Unfortunately, this information is impossible to obtain in most cases because companies consider their technologies proprietary and want to keep

[2] When it downloads page c, it does not remember whether the link to g has been already sent.

[3] According to our experiments, about 90% of the links in a page point to pages in the same site on average.

[4] While the sizes of individual Web site vary, the sizes of partitions are similar, because each partition contains many Web sites and their *average* sizes are similar among partitions.

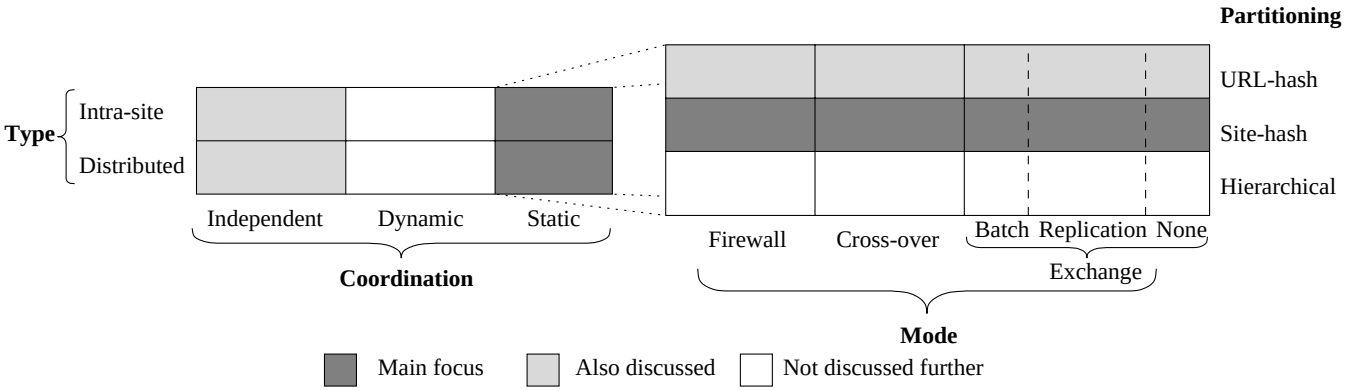


Figure 3: Summary of the options discussed

them secret. The only two crawler designs that we know of are the prototype Google crawler [23] (when it was developed at Stanford) and the Mercator crawler [14] at Compaq. The prototype google crawler used the intra-site, static and site-hash based scheme and ran in exchange mode [23]. The Mercator crawler uses the site-based hashing scheme.

4. EVALUATION MODELS

In this section, we define metrics that will let us quantify the advantages or disadvantages of different parallel crawling schemes. These metrics will be used later in our experiments.

1. **Overlap:** When multiple C-proc's are downloading Web pages simultaneously, it is possible that different C-proc's download the same page multiple times. Multiple downloads of the same page are clearly undesirable.

   More precisely, we define the *overlap* of downloaded pages as $\frac{N-I}{I}$. Here, N represents the *total* number of pages downloaded by the overall crawler, and I represents the number of *unique* pages downloaded, again, by the overall crawler. Thus, the goal of a parallel crawler is to minimize the overlap.

   Note that a parallel crawler does not have an overlap problem, if it is based on the firewall mode (Section 3 Item 1) or the exchange mode (Section 3 Item 3). In these modes, a C-proc downloads pages only within its own partition, so the overlap is always zero.

2. **Coverage:** When multiple C-proc's run independently, it is possible that they may not download all pages that they have to. In particular, a crawler based on the firewall mode (Section 3 Item 1) may have this problem, because its C-proc's do not follow inter-partition links nor exchange the links with others.

   To formalize this notion, we define the *coverage* of downloaded pages as $\frac{I}{U}$, where U represents the total number of pages that the overall crawler has to download, and I is the number of unique pages downloaded by the overall crawler. For example, in Figure 2, if C_1 downloaded pages a, b and c, and if C_2 downloaded pages f through i, the coverage of the overall crawler is $\frac{7}{9} = 0.77$, because it downloaded 7 pages out of 9.

3. **Quality:** Often, crawlers cannot download the whole Web, and thus they try to download an "important" or "relevant" section of the Web. For example, a crawler may have storage space only for 1 million pages and may want to download the "most important" 1 million pages. To implement this policy, a crawler needs a notion of "importance" of pages, often called an *importance metric* [8].

   For example, let us assume that the crawler uses *backlink count* as its importance metric. That is, the crawler considers a page p important when a lot of other pages point to it. Then the goal of the crawler is to download the most highly-linked 1 million pages. To achieve this goal, a single-process crawler may use the following method [8]: The crawler constantly keeps track of how many backlinks each page has from the pages that it has already downloaded, and first visits the page with the highest backlink count. Clearly, the pages downloaded in this way may not be the top 1 million pages, because the page selection is not based on the entire Web, only on what has been seen so far. Thus, we may formalize the notion of "quality" of downloaded pages as follows [8]:

   First, we assume a hypothetical *oracle crawler*, which knows the exact importance of every page under a certain importance metric. We assume that the oracle crawler downloads the most important N pages in total, and use P_N to represent that set of N pages. We also use A_N to represent the set of N pages that an *actual crawler* would download, which would not be necessarily the same as P_N. Then we define $\frac{|A_N \cap P_N|}{|P_N|}$ as the *quality* of downloaded pages by the actual crawler. Under this definition, the quality represents the fraction of the true top N pages that are downloaded by the crawler.

   Note that the quality of a parallel crawler may be worse than that of a single-process crawler, because many importance metrics depend on the global structure of the Web (e.g., backlink count). That is, each C-proc in a parallel crawler may know only the pages that are downloaded *by itself*, and thus have less information on page importance than a single-process crawler does. On the other hand, a single-process crawler knows *all* pages it has downloaded so far. Therefore, a

C-proc in a parallel crawler may make a worse crawling decision than a single-process crawler.

In order to avoid this quality problem, C-proc's need to periodically exchange information on page importance. For example, if the backlink count is the importance metric, a C-proc may periodically notify other C-proc's of how many pages in its partition have links to pages in other partitions.

Note that this backlink exchange can be naturally incorporated in an exchange mode crawler (Section 3 Item 3). In this mode, crawling processes exchange inter-partition URLs periodically, so a C-proc can simply count how many inter-partition links it receives from other C-proc's, to count backlinks originating in other partitions. More precisely, if the crawler uses the batch communication technique (Section 3.1 Item 1), process C_1 would send a message like `[http://cnn.com/index.html, 3]` to C_2, to notify that C_1 has seen 3 links to the page in the current batch.[5] On receipt of this message, C_2 then increases the backlink count for the page by 3 to reflect the inter-partition links. By incorporating this scheme, we believe that the quality of the exchange mode will be better than that of the firewall mode or the cross-over mode.

However, note that the quality of an exchange mode crawler may vary depending on how often it exchanges backlink messages. For instance, if crawling processes exchange backlink messages after every page download, they will have essentially the same backlink information as a single-process crawler does. (They know backlink counts from *all* pages that have been downloaded.) Therefore, the quality of the downloaded pages would be virtually the same as that of a single-process crawler. In contrast, if C-proc's rarely exchange backlink messages, they do not have "accurate" backlink counts from downloaded pages, so they may make poor crawling decisions, resulting in poor quality. Later, we will study how often C-proc's should exchange backlink messages in order to maximize the quality.

4. **Communication overhead:** The C-proc's in a parallel crawler need to exchange messages to coordinate their work. In particular, C-proc's based on the exchange mode (Section 3 Item 3) swap their inter-partition URLs periodically. To quantify how much communication is required for this exchange, we define *communication overhead* as the average number of inter-partition URLs exchanged per downloaded page. For example, if a parallel crawler has downloaded 1,000 pages in total and if its C-proc's have exchanged 3,000 inter-partition URLs, its communication overhead is $3,000/1,000 = 3$. Note that a crawler based on the the firewall and the cross-over mode do not have any communication overhead, because they do not exchange any inter-partition URLs.

In Table 1, we compare the relative merits of the three crawling modes (Section 3 Items 1–3). In the table, "Good" means that the mode is expected to perform relatively well for that metric, and "Bad" means that it may perform worse compared to other modes. For instance, the firewall mode does not exchange any inter-partition URLs (Communication: Good) and downloads pages only once (Overlap: Good), but it may not download every page (Coverage: Bad). Also, because C-proc's do not exchange inter-partition URLs, the downloaded pages may be of low quality than those of an exchange mode crawler. Later, we will examine these issues more quantitatively through experiments based on real Web data.

[5]If the C-proc's send inter-partition URLs incrementally after every page, the C-proc's can send the URL only, and other C-proc's can simply count these URLs.

Mode	*Coverage*	*Overlap*	*Quality*	*Communication*
Firewall	Bad	Good	Bad	Good
Cross-over	Good	Bad	Bad	Good
Exchange	Good	Good	Good	Bad

Table 1: Comparison of three crawling modes

5. DESCRIPTION OF DATASET

We have discussed various issues related to a parallel crawler and identified multiple alternatives for its architecture. In the remainder of this paper, we quantitatively study these issues through experiments conducted on real Web data.

In all of the following experiments, we used 40 millions Web pages in our Stanford WebBase repository. Because the property of this dataset may significantly impact the result of our experiments, readers might be interested in how we collected these pages.

We downloaded the pages using our Stanford WebBase crawler in December 1999 in the period of 2 weeks. In downloading the pages, the WebBase crawler started with the URLs listed in Open Directory (`http://www.dmoz.org`), and followed links. We decided to use the Open Directory URLs as seed URLs, because these pages are the ones that are considered "important" by its maintainers. In addition, some of our local WebBase users were keenly interested in the Open Directory pages and explicitly requested that we cover them. The total number of URLs in the Open Directory was around 1 million at that time. Then conceptually, the WebBase crawler downloaded all these pages, extracted URLs within the downloaded pages, and followed links in a breadth-first manner. (The WebBase crawler uses various techniques to expedite and prioritize crawling process, but we believe these optimizations do not affect the final dataset significantly.)

Because our dataset was downloaded by a crawler in a particular way, our dataset may not correctly represent the *actual* Web as it is. In particular, our dataset may be biased towards more "popular pages," because we started from Open Directory pages. Also, our dataset does not cover the pages that are accessible *only through* a query interface. For example, our crawler did not download the pages generated by a keyword-based search engine, because it did not try to "guess" appropriate keywords to fill in. However, we emphasize that many of dynamically-generated pages were still downloaded by our crawler. For example, the pages on Amazon web site (`http://amazon.com`) are dynamically generated, but we could still download most of the pages on the site by following links.

In summary, our dataset may not necessarily reflect the actual image of the Web, but we believe it represents the image that a *crawler would see* in its crawl. In most cases,

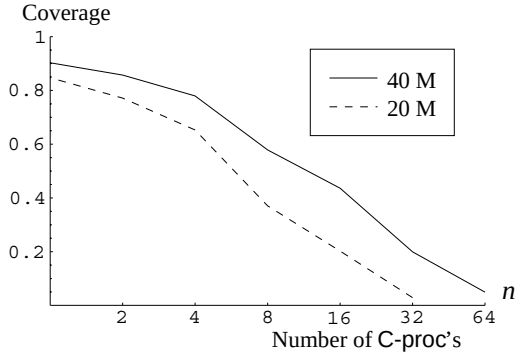


Figure 4: Number of processes vs. Coverage

crawlers are mainly interested in downloading "popular" or "important" pages, and they download pages by following links, which is what we did for our data collection.

Our dataset is relatively "small" (40 million pages) compared to the full Web, but keep in mind that using a significantly larger dataset would have made many of our experiments prohibitively expensive. As we will see, each of the graphs we present study multiple configurations, and for each configuration, multiple crawler runs were made to obtain statistically valid data points. Each run involves simulating how one or more C-proc would visit the 40 million pages. Such detailed simulations are inherently very time consuming. As we present our results, we will return to the dataset size issue at various points, and discuss whether a larger dataset would have changed our conclusions. In one case where conclusions could be impacted, we run an additional experiment with 20 million pages, to understand the impact of a growing dataset.

6. FIREWALL MODE AND COVERAGE

A firewall mode crawler (Section 3 Item 1) has minimal communication overhead, but it may have coverage and quality problems (Section 4). In this section, we quantitatively study the effectiveness of a firewall mode crawler using the 40 million pages in our repository. In particular, we estimate the coverage (Section 4 Item 2) of a firewall mode crawler when it employs n C-proc's in parallel. (We discuss the quality issue of a parallel crawler later.)

In our experiments, we considered the 40 million pages within our WebBase repository as the entire Web, and we used site-hash based partitioning (Section 3.2 Item 2). As the seed URLs, each C-proc was given 5 random URLs from its own partition, so $5n$ seed URLs were used in total by the overall crawler. (We discuss the effect of the number of seed URLs shortly.) Since the crawler ran in firewall mode, C-proc's followed only *intra-partition* links, not inter-partition links. Under these settings, we let the C-proc's run until they ran out of URLs. After this simulated crawling, we measured the overall coverage of the crawler. We performed these experiments with $5n$ *random* seed URLS and repeated the experiments multiple times with different seed URLs. In all of the runs, the results were essentially the same.

In Figure 4, we summarize the results from the experiments. The horizontal axis represents n, the number of parallel C-proc's, and the vertical axis shows the coverage of the overall crawler for the given experiment. The solid line in the graph is the result from the 40M page experiment.

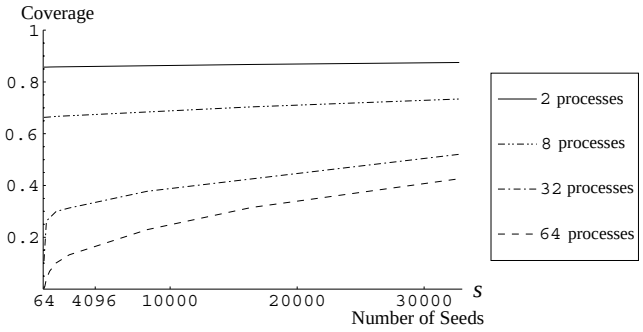


Figure 5: Number of seed URLs vs. Coverage

(We explain the dotted line later on.) Note that the coverage is only 0.9 even when $n = 1$ (a single-process). This result is because the crawler in our experiment started with only 5 URLs, while the actual dataset was collected with 1 million seed URLs. Thus, some of the 40 million pages were unreachable from the 5 seed URLs.

From the figure it is clear that the coverage decreases as the number of processes increases. This trend is because the number of inter-partition links increases as the Web is split into smaller partitions, and thus more pages are reachable only through inter-partition links.

From this result we can see that we may run a crawler in a firewall mode without much decrease in coverage with fewer than 4 C-proc's. For example, for the 4 process case, the coverage decreases only 10% from the single-process case. At the same time, we can also see that the firewall mode crawler becomes quite ineffective with a large number of C-proc's. Less than 10% of the Web can be downloaded when 64 C-proc's run together, each starting with 5 seed URLs.

Clearly, coverage may depend on the number of seed URLs that each C-proc starts with. To study this issue, we also ran experiments varying the number of seed URLs, s, and we show the results in Figure 5. The horizontal axis in the graph represents s, the *total* number of seed URLs that the overall crawler used, and the vertical axis shows the coverage for that experiment. For example, when $s = 128$, the overall crawler used 128 total seed URLs, each C-proc starting with 2 seed URLs when 64 C-proc's ran in parallel. We performed the experiments for 2, 8, 32, 64 C-proc cases and plotted their coverage values. From this figure, we can observe the following trends:

- *When a large number of* C-proc*'s run in parallel, (e.g.,* 32 *or* 64*), the total number of seed URLs affects the coverage very significantly.* For example, when 64 processes run in parallel the coverage value jumps from 0.4% to 10% if the number of seed URLs increases from 64 to 1024.
- *When only a small number of processes run in parallel (e.g.,* 2 *or* 8*), coverage is not significantly affected by the number of seed URLs.* While coverage increases slightly as s increases, the improvement is marginal.

Based on these results, we draw the following conclusions:

1. When a relatively small number of C-proc's are running in parallel, a crawler using the firewall mode provides good coverage. In this case, the crawler may start with only a small number of seed URLs, because coverage is not much affected by the number of seed URLs.

2. The firewall mode is not a good choice if the crawler wants to download every single page on the Web. The crawler may miss some portion of the Web, particularly when it runs many C-proc's in parallel.

Our results in this section are based on a 40 million page dataset, so it is important to consider how coverage might change with a different dataset, or equivalently, how it might change as the Web grows or evolves. Unfortunately, it is difficult to predict how the Web will grow. On one hand, if all "newly created" pages are well connected to existing pages at their creation site, then coverage will increase. On the other hand, if new pages tend to form disconnected groups, the overall coverage will decrease. Depending on how the Web grows, coverage could go either way.

As a preliminary study of the growth issue, we conducted the same experiments with a subset of 20M pages and measured how the coverage changes.[6] We first randomly selected a half of the *sites* in our dataset, and ran the experiments using only the pages from those sites. Thus, one can roughly view the smaller dataset as a smaller Web, that then "over time" doubled its number of sites to yield the second dataset. The dotted line in Figure 4 shows the results from the 20M-page experiments. From the graph we can see that as "our Web doubles in size," one can double the number of C-proc's and retain roughly the same coverage. That is, the new sites can be visited by new C-proc's without significantly changing the coverage they obtain. If the growth did not exclusively come from new sites, then one should not quite double the number of C-proc's each time the Web doubles in size, to retain the same coverage.

EXAMPLE 1. *(Generic search engine) To illustrate how our results could guide the design of a parallel crawler, consider the following example. Assume that to operate a Web search engine, we need to download 1 billion pages*[7] *in one month. Each machine that we plan to run our C-proc's on has 10 Mbps link to the Internet, and we can use as many machines as we want.*

Given that the average size of a web page is around 10K bytes, we roughly need to download $10^4 \times 10^9 = 10^{13}$ *bytes in one month. This download rate corresponds to 34 Mbps, and we need 4 machines (thus 4 C-proc's) to obtain the rate. If we want to be conservative, we can use the results of our 40M-page experiment (Figure 4), and estimate that the coverage will be at least* 0.8 *with 4 C-proc's. Therefore, in this scenario, the firewall mode may be good enough, unless it is very important to download the "entire" Web.*

EXAMPLE 2. *(High freshness) As a second example, let us now assume that we have strong "freshness" requirement on the 1 billion pages and need to revisit every page once every week, not once every month. This new scenario requires approximately 140 Mbps for page download, and we need to run* 14 C-proc*'s. In this case, the coverage of the overall crawler decreases to less than* 0.5 *according to Figure 4. Of course, the coverage could be larger than our conservative estimate, but to be safe one would probably want to consider using a crawler mode different than the firewall mode.*

[6]We recently completed a crawl of about 130M pages, and we are also running the experiments on this new dataset. (Each run takes over a week to complete.) We will include the results in the final version of this paper.

[7]Currently the Web is estimated to have around 1 billion pages.

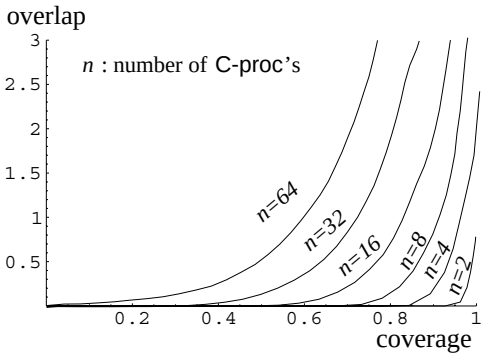


Figure 6: Coverage vs. Overlap for a cross-over mode crawler

7. CROSS-OVER MODE AND OVERLAP

In this section, we study the effectiveness of a cross-over mode crawler (Section 3, Item 2). A cross-over crawler may yield improved coverage of the Web, since it follows inter-partition links when a C-proc runs out of URLs in its own partition. However, this mode incurs overlap in downloaded pages (Section 4, Item 1), because a page can be downloaded by multiple C-proc's. Therefore, the crawler increases its coverage at the expense of overlap in the downloaded pages.

In Figure 6, we show the relationship between the coverage and the overlap of a cross-over mode crawler obtained from the following experiments. We partitioned the 40M pages using site-hash partitioning and assigned them to n C-proc's. Each of the n C-proc's then was given 5 random seed URLs from its partition and followed links in the cross-over mode. During this experiment, we measured how much overlap the overall crawler incurred when its coverage reached various points. The horizontal axis in the graph shows the coverage at a particular time and the vertical axis shows the overlap at the given coverage. We performed the experiments for $n = 2, 4, \ldots, 64$.

Note that in most cases the overlap stays at zero until the coverage becomes relatively large. For example, when $n = 16$, the overlap is zero until coverage reaches 0.5. We can understand this result by looking at the graph in Figure 4. According to that graph, a crawler with 16 C-proc's can cover around 50% of the Web by following only intra-partition links. Therefore, even a cross-over mode crawler will follow only intra-partition links until its coverage reaches that point. Only after that, each C-proc starts to follow inter-partition links, thus increasing the overlap. For this reason, we believe that the overlap would have been much worse in the beginning of the crawl, if we adopted the independent model (Section). By applying the partitioning scheme to C-proc's, we make each C-proc stay in its own partition in the beginning and suppress the overlap as long as possible.

While the crawler in the cross-over mode is much better than one based on the independent model, it is clear that the cross-over crawler still incurs quite significant overlap. For example, when 4 C-proc's run in parallel in the cross-over mode, the overlap becomes almost 2.5 to obtain coverage close to 1. For this reason, we do not recommend the cross-over mode, unless it is absolutely necessary to download every page without any communication between C-proc's.

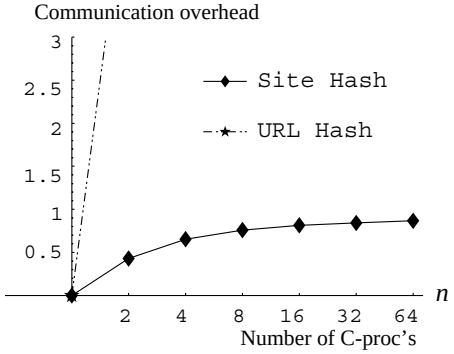


Figure 7: Number of crawling processes vs. Number of URLs exchanged per page

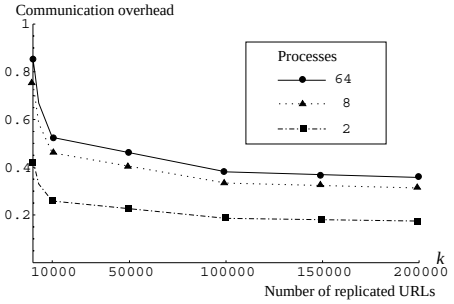


Figure 8: Number of replicated URLs vs. Number of URLs exchanged per page

8. EXCHANGE MODE AND COMMUNICATION

To avoid the overlap and coverage problems, an exchange mode crawler (Section 3, Item 3) constantly exchanges inter-partition URLs between C-proc's. In this section, we study the communication overhead (Section 4, Item 4) of an exchange mode crawler and how much we can reduce it by replicating the most popular k URLs. For now, let us assume that a C-proc *immediately* transfers inter-peartition URLs. (We will discuss batch communication later when we discuss the quality of a parallel crawler.)

In the experiments, again, we split the 40 million pages into n partitions based on site-hash values and ran n C-proc's in the exchange mode. At the end of the crawl, we measured how many URLs had been exchanged during the crawl. We show the results in Figure 7. In the figure, the horizontal axis represents the number of parallel C-proc's, n, and the vertical axis shows the communication overhead (the average number of URLs transferred per page). For comparison purposes, the figure also shows the overhead for a URL-hash based scheme, although the curve is clipped at the top because of its large overhead values.

To explain the graph, we first note that an average page has 10 out-links, and about 9 of them point to pages in the *same* site. Therefore, the 9 links are internally followed by a C-proc under site-hash partitioning. Only the remaining 1 link points to a page in a different site and may be exchanged between processes. Figure 7 indicates that this URL exchange increases with the number of processes. For example, the C-proc's exchanged 0.4 URLs per page when 2 processes ran, while they exchanged 0.8 URLs per page when 16 processes ran. Based on the graph, we draw the following conclusions:

- *The site-hash based partitioning scheme significantly reduces communication overhead,* compared to the URL-hash based scheme. We need to transfer only *up to one link per page* (or 10% of the links), which is significantly smaller than the URL-hash based scheme. For example, when we ran 2 C-proc's using the URL-hash based scheme the crawler exchanged 5 links per page under the URL-hash based scheme, which was significantly larger than 0.5 links per page under the site-hash based scheme.
- *The network bandwidth used for the URL exchange is relatively small, compared to the actual page download bandwidth.* Under the site-hash based scheme, at most 1 URL will be exchanged per page, which is about 40 bytes.[8] Given that the average size of a Web page is 10 KB, the URL exchange consumes less than $40/10\text{K} = 0.4\%$ of the total network bandwidth.
- However, *the overhead of the URL exchange on the overall system can be quite significant.* The processes need to exchange up to one message per page, and the message has to go through the TCP/IP network stack at the sender *and* the receiver. Thus it is copied to and from kernel space twice, incurring two context switches between the kernel and the user mode. Since these operations pose significant overhead even if the message size is small, the overall overhead can be important if the processes exchange one message per every downloaded page.

In order to study how much we can reduce this overhead by replication (Section 3.1, Item 2), in Figure 8 we show the communication overhead when we replicate the top k popular URLs. In the figure, the horizontal axis shows the number of replicated URLs, k, and the vertical axis shows the communication overhead.

Remember that in a real-world scenario, we would identify the most popular URLs based on the image of the Web from a previous crawl. However, in our experiment, we identified the top k URLs based on the 40 million pages in our *current* WebBase repository, which was also used for our experiments. Therefore, there had been no change over time, and the replicated URLs were *exactly* the most popular ones in the repository. For this reason, in a real-world scenario, the actual communication overhead might be slightly worse than what our results show.

From Figure 8 it is clear that we can significantly reduce the communication overhead by replicating a relatively small number of URLs. For example, when 64 C-proc's run in parallel, the overhead reduces from 0.86 URLs/page to 0.52 URLs/page (40% reduction) when we replicate only $10,000$ URLs. From the figure, we can also see that this reduction diminishes as the number of replicated URLs increases. For example, when we replicate $100,000$ URLs, we can get about 51% reduction (from 0.86 to 0.42, 64 process case), while we can get only 53% reduction when we replicate $200,000$ URLs. Based on this result, we recommend replicating between $10,000$ to $100,000$ UCLs in each C-proc, in order to minimize the communication overhead while maintaining a low replication overhead.

[8]In our estimation, an average URL was about 40 bytes long.

While our experiments were based on 40 million pages, the results will be similar, even if we were to run our experiments on a larger dataset. Note that our results depend on the fraction of links pointing to the top k URLs. We believe that the links in our experiments are good samples for the Web, because the pages were downloaded from the Web and the links in the pages were created by independent authors. Therefore, a similar fraction of links would point to the top k URLs, even if we download more pages, or equivalently, if the Web grows over time.

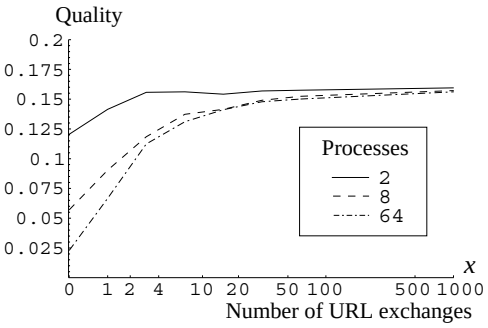
9. QUALITY AND BATCH COMMUNICATION

As we discussed, the quality (Section 4, Item 3) of a parallel crawler can be worse than that of a single-process crawler, because each C-proc may make crawling decisions solely based on the information collected within its own partition. We now study this quality issue. In the discussion we also study the impact of the batch communication technique (Section 3.1 Item 1) on quality.

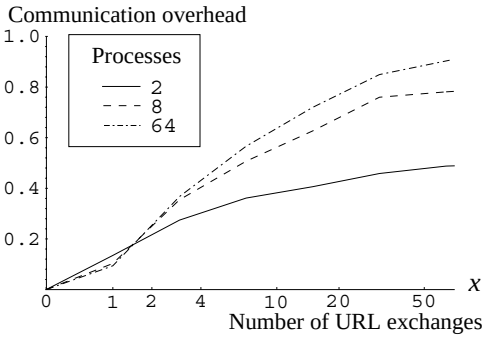
Throughout the experiments in this section, we assume that the crawler uses the *number of backlinks* to page p as the importance of p, or $I(p)$. That is, if 1000 pages on the Web have links to page p, the importance of p is $I(p) = 1000$. Clearly, there exist many other ways to define the importance of a page, but we use this metric because it (or its variations) is being used by some existing search engines [23, 12]. Also, note that this metric depends on the global structure of the Web. If we use an importance metric that solely depends on a page itself, not on the global structure of the Web, the quality of a parallel crawler will be essentially the same as that of a single crawler, because each C-proc in a parallel crawler can make good decisions based on the pages that it has downloaded.

Under the backlink metric, each C-proc in our experiments counted how many backlinks a page has from the downloaded pages and visited the page with the most backlinks first. Remember that the C-proc's need to periodically exchange messages to inform others of the inter-partition backlinks. Depending on how often they exchange messages, the quality of the downloaded pages will differ. For example, if C-proc's *never* exchange messages, the quality will be the same as that of a firewall mode crawler, and if they exchange messages after every downloaded page, the quality will be similar to that of a single-process crawler.

To study these issues, we compared the quality of the downloaded pages when C-proc's exchanged backlink messages at various intervals and we show the results in Figures 9(a), 10(a) and 11(a). Each graph shows the quality achieved by the overall crawler when it downloaded a total of 500K, 2M, and 8M pages, respectively. The horizontal axis in the graphs represents *the total number of URL exchanges* during a crawl, x, and the vertical axis shows the quality for the given experiment. For example, when $x = 1$, the C-proc's exchanged backlink count information *only once* in the middle of the crawl. Therefore, the case when $x = 0$ represents the quality of a firewall mode crawler, and the case when $x \to \infty$ shows the quality a single-process crawler. In Figure 9(b), 10(b) and 11(b), we also show the communication overhead (Section 4, Item 4); that is, the average number of `[URL, backlink count]` pairs exchanged per a downloaded page.



(a) URL exchange vs. Quality

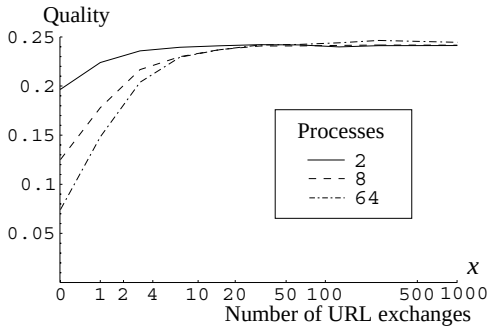


(b) URL exchange vs. Communication

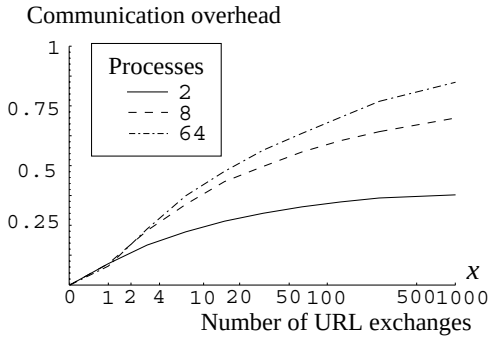
Figure 9: Crawlers downloaded 500K pages (1.2% of 40M)

From these figures, we can observe the following trends:

- *As the number of crawling processes increases, the quality of downloaded pages becomes worse, unless they exchange backlink messages often.* For example, in Figure 9(a), the quality achieved by a 2 process crawler (0.12) is significantly higher than that of a 64 process crawler (0.025) in the firewall mode ($x = 0$). Again, this result is because each C-proc learns less about the global backlink counts when the Web is split into smaller parts.
- *The quality of the firewall mode crawler ($x = 0$) is significantly worse than that of the single-process crawler ($x \to \infty$) when the crawler downloads a relatively small fraction of the pages (Figure 9(a) and 10(a)).* However, the difference is not very significant when the crawler downloads a relatively large fraction (Figure 11 (a)). In other experiments, when the crawler downloaded more than 50% of the pages, the difference was almost negligible in any case. (Due to space limitations, we do not show the graphs.) Intuitively, this result makes sense because quality is an important issue only when the crawler downloads a small portion of the Web. (If the crawler will visit all pages anyway, quality is not relevant.)
- *The communication overhead does not increase linearly as the number of URL exchange increases.* The graphs in Figure 9(b), 10(b) and 11(b) are not straight

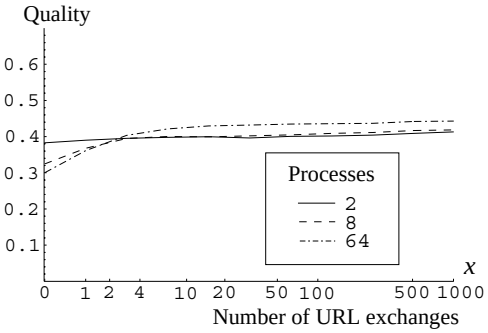


(a) URL exchange vs. Quality

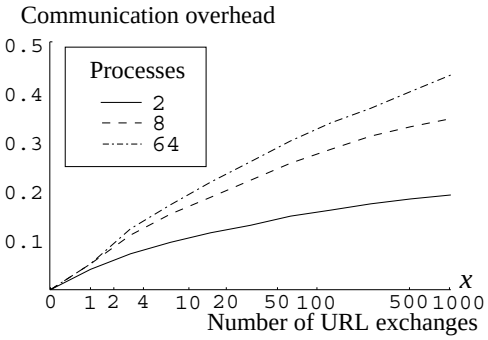


(b) URL exchange vs. Communication

Figure 10: Crawlers downloaded 2M pages (5% of 40M)



(a) URL exchange vs. Quality



(b) URL exchange vs. Communication

Figure 11: Crawlers downloaded 8M pages (20% of 40M)

lines. This result is because a popular URL appear multiple times between backlink exchanges. Therefore, a popular URL can be transferred as *one* entry (URL and its backlink count) in the exchange, even if it appeared multiple times. This reduction increases as C-proc's exchange backlink messages less frequently.

- *One does not need a large number of URL exchanges to achieve high quality.* Through multiple experiments, we tried to identify how often C-proc's should exchange backlink messages to achieve the highest quality value. From these experiments, we found that a parallel crawler can get the highest quality values even if the processes communicate *less than* 100 *times* during a crawl.

We use the following example to illustrate how one can use the results of our experiments.

EXAMPLE 3. *(Medium-Scale Search Engine) Say we plan to operate a medium-scale search engine, and we want to maintain about* 20% *of the Web (200 M pages) in our index. Our plan is to refresh the index once a month. The machines that we can use have individual T1 links (1.5 Mbps) to the Internet.*

In order to update the index once a month, we need about 6.2 Mbps download bandwidth, so we have to run at least 5 C-proc*'s on 5 machines. According to Figure 11(a) (*20% *download case), we can achieve the highest quality if the* C-proc*'s exchange backlink messages 10 times during a crawl when* 8 *processes run in parallel. (We use the 8 process case because it is the closest number to 5). Also, from Figure 11(b), we can see that when* C-proc*'s exchange messages 10 times during a crawl they need to exchange fewer than* $0.17 \times 200\text{M} = 34\text{M}$ `[URL, backlink count]` *pairs in total. Therefore, the total network bandwidth used by the backlink exchange is only* $(34\text{M} \cdot 40)/(200\text{M} \cdot 10\text{K}) \approx 0.06\%$ *of the bandwidth used by actual page downloads. Also, since the exchange happens only 10 times during a crawl, the context-switch overhead for message transfers (discussed in Section 8) is minimal.*

Note that in this scenario we need to exchange 10 backlink messages in one month or one message every three days. Therefore, even if the connection between C-proc*'s is unreliable or sporadic, we can still use the exchange mode without any problem.*

10. CONCLUSION

Crawlers are being used more and more often to collect Web data for search engine, caches, and data mining. As the size of the Web grows, it becomes increasingly important to use parallel crawlers. Unfortunately, almost nothing is known (at least in the open literature) about options for parallelizing crawlers and their performance. Our paper addresses this shortcoming by presenting several architectures and strategies for parallel crawlers, and by studying their performance. We believe that our paper offers some useful guidelines for crawler designers, helping them for example,

select the right number of crawling processes, or select the proper inter-process coordination scheme.

In summary, the main conclusions of our study were the following:

- When a small number of crawling processes run in parallel (in our experiment, 4 or fewer), the firewall mode provides good coverage. Given that firewall mode crawlers can run totally independently and are easy to implement, we believe that it is a good option to consider. The cases when the firewall mode might not be appropriate are:
  1. when we need to run more than 4 crawling processes or
  2. when we download only a small subset of the Web and the quality of the downloaded pages is important.
- A crawler based on the exchange mode consumes small network bandwidth for URL exchanges (less than 1% of the network bandwidth). It can also minimize other overheads by adopting the batch communication technique. In our experiments, the crawler could maximize the quality of the downloaded pages, even if it exchanged backlink messages fewer than 100 times during a crawl.
- By replicating between $10,000$ and $100,000$ popular URLs, we can reduce the communication overhead by roughly 40%. Replicating more URLs does not significantly reduce the overhead.

11. REFERENCES

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, 1995.

[2] A. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(509), 1999.

[3] A. Z. Broder, S. R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the web. In *Proceedings of the Ninth World-Wide Web Conference*, 2000.

[4] M. Burner. Crawling towards eterneity: Building an archive of the world wide web. *Web Techniques Magazine*, 2(5), May 1998.

[5] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *The 8th International World Wide Web Conference*, 1999.

[6] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, 2000.

[7] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the 2000 ACM SIGMOD*, 2000.

[8] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through URL ordering. *Computers networks and ISDN systems*, 30:161–172, 1998.

[9] E. Coffman, Jr., Z. Liu, and R. R. Weber. Optimal robot scheduling for web search engines. Technical report, INRIA, 1997.

[10] M. Diligenti, F. M. Coetzee, S. Lawrence, C. L. Giles, and M. Gori. Focused crawling using context graphs. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, 2000.

[11] D. Eichmann. The RBSE spider: Balancing effective search against web load. In *Proceedings of the First World-Wide Web Conference*, 1994.

[12] Google Inc. `http://www.google.com`.

[13] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *Word Wide Web*, 2(4):219–229, December 1999.

[14] A. Heydon and M. Najork. High-performance web crawling. Technical report, SRC Research Report, 173, Compaq Systems Research Center, September 2001.

[15] D. Hirschberg. Parallel algorithms for the transitive closure and the connected component problem. In *Proceedings of the 8th Annual ACM Symposium on the Theory of Computing*, 1976.

[16] M. Koster. Robots in the web: threat or treat? *ConneXions*, 4(4), April 1995.

[17] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–357, November 1989.

[18] O. A. McBryan. GENVL and WWWW: Tools for taming the web. In *Proceedings of the First World-Wide Web Conference*, 1994.

[19] R. C. Miller and K. Bharat. SPHINX: a framework for creating personal, site-specific web crawlers. In *Proceedings of the Seventh World-Wide Web Conference*, 1998.

[20] D. S. MiloJicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[21] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of ACM*, 29:642–667, July 1982.

[22] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[23] L. Page and S. Brin. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh World-Wide Web Conference*, 1998.

[24] B. Pinkerton. Finding what people want: Experiences with the web crawler. In *Proceedings of the Second World-Wide Web Conference*, 1994.

[25] M. J. Quinn and N. Deo. Parallel graph algorithms. *ACM Computing Surveys*, 16(3), September 1984.

[26] Robots exclusion protocol. `http://info.webcrawler.com/mak/projects/robots/exclusion.html`.

[27] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[28] A. S. Tanenbaum and R. V. Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4), December 1985.

[29] G. K. Zipf. *Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology*. Addison-Wesley, 1949.