A Full-Coverage Two-Level URL Duplication Checking Method for a High-Speed Parallel Web Crawler*

ARJUMAND YOUNUS, KYU-YOUNG WHANG, HYUK-YOON KWON
AND YEON-MI YEO
Department of Computer Science
Korea Advanced Institute of Science and Technology
Daejeon 305-701 Korea
E-mail: {arjumand; kywhang; hykwon; ymyeo}@mozart.kaist.ac.kr

For efficient large-scale Web crawlers, URL duplication checking is an important technique since it is a significant bottleneck. In this paper, we propose a new URL duplication checking technique for a parallel Web crawler; we call it full-coverage two level URL duplication checking (full-coverage-2L-UDC). Full-coverage-2L-UDC provides efficient URL duplication checking while ensuring maximum coverage. First, we propose two-level URL duplication checking (2L-UDC). It provides efficiency in URL duplication checking by communicating at the Web site level rather than at the Web page level. Second, we present a solution for the so-called coverage problem, which is directly related to the recall of the search engine. It is the first solution for the coverage problem in the centralized parallel architecture. Third, we propose an architecture, FC2L-UDCbot, for a centralized parallel crawler using full-coverage-2L-UDC. We build a seven-agent FC2L-UDCbot for extensive experiments. We show that the crawling speed of FC2L-UDCbot is approximately proportional to the number of agents (i.e., FC2L-UDCbot is faster than a single-machine crawler by 6.9 times). Full-coverage-2L-UDC allows FC2L-UDCbot to be scalable to the number of agents since it effectively deals with the overheads incurred in a parallel environment. Through an in-depth analysis, we construct a cost model for estimating the crawling speed of a scaled-up crawler. Using the model, we show that FC2L-UDCbot can crawl Google-scale Web pages within several days using dozens of agents.

Keywords: centralized parallel crawler, two-level URL duplication checking, coverage problem, high-speed parallel Web crawler, cost model

1. INTRODUCTION

World Wide Web is a significant information gathering medium. However, due to information explosion [1] on this medium, the task of finding information has been becoming ever more difficult. The importance of a Web search engine is thus highlighted as a tool that makes it easier to find the necessary information from the Web [2]. A standard Web search engine basically consists of two main parts: the first comprises a Web crawler that is responsible for fetching Web pages and an indexer that builds the index; the second is a query processor that processes a stream of queries [3].

A Web crawler starts with some given seed URLs, downloads the Web pages of the URLs, and navigates following all the hyperlinks found in the downloaded Web pages. In this way, a Web crawler may find new Web sites as a result of navigation. This pro-
cess continues until a certain termination condition (e.g., the maximum number of downloaded Web sites or Web pages) is satisfied.

It is fairly easy to build a slow crawler that downloads a few pages per second for a short period of time. However, the challenge is in the design of a high-speed crawler that can download tens of billions of pages over a few days [4]. There have been a number of techniques proposed for a high-speed crawler. Parallelization is important since there is a strong need to “parallelize the crawling process in order to finish downloading pages in a reasonable amount of time” [5]. We review the techniques for parallel Web crawling. Brin et al. [6] have proposed the Google crawler, which is a centralized parallel crawler that follows the client-server model. Here, the server is the coordinator, and the clients are agents. Cho et al. [5, 7] have proposed the Stanford WebBase crawler, which is a decentralized parallel crawler that follows the client-to-client model. The centralized architecture is widely used for parallel Web crawling [8, 9]; thus, we adopt this.

There are many important issues for crawlers such as politeness policy, spam control, and fault tolerance. However, URL duplication checking is a bottleneck for efficient large-scale Web crawling [4, 10]. Thus, we focus on URL duplication checking in this paper excluding the effects of the other issues to the performance of Web crawlers. For efficient URL duplication checking, Heydon et al. [11] have proposed a hybrid approach that utilizes both disk and memory in order to achieve scalability. Lee et al. [10] have proposed a technique named Disk Repository with Update Management (DRUM). DRUM obtains efficiency by using deferred, batch URL duplication checking based on the hybrid approach. DRUM is currently the most efficient URL duplication checking method for a single-machine crawler.

Unlike in a single-machine crawler, parallelization of crawlers causes other issues for efficient URL duplication checking: (1) efficient communication between the coordinator and agents and (2) the coverage problem [5], which is directly related to the recall of the search engine.

In this paper, we propose a new URL duplication checking method for a centralized parallel Web crawler. We call it full-coverage two-level URL duplication checking (full-coverage-2L-UDC). Full-coverage-2L-UDC is an efficient URL duplication checking method that ensures maximum coverage. First, we propose two-level URL duplication checking (2L-UDC). It enhances efficiency since it reduces the communication cost between the coordinator and agents by communicating at the Web site level rather than at the Web page level. Second, we observe that the coverage problem occurs in 2L-UDC and present a solution for it. It is the first to solve the coverage problem in the centralized architecture. Third, we propose an architecture of the centralized parallel crawler using full-coverage-2L-UDC, which we call FC2L-UDCbot. We build a seven-agent FC2L-UDCbot for extensive experiments. We show that the crawling speed of FC2L-UDCbot is approximately proportional to the number of agents (i.e., FC2L-UDCbot is faster than a single-machine crawler by 6.9 times). Full-coverage-2L-UDC allows FC2L-UDCbot to be scalable to the number of agents since it effectively deals with the overheads incurred in a parallel environment.

We construct a performance model for estimating the crawling speed of a scaled-up FC2L-UDCbot with a larger number of agents. To obtain realistic experimental results in the absence of a physical high-bandwidth network (e.g., 10Gbits/sec), we propose a technique for emulating the high-bandwidth network using an emulated World Wide
Web on local disks. Then, we build a seven-agent FC2L-UDCbot for extensive experiments. Through an in-depth analysis, we identify the types of overheads that are incurred in FC2L-UDCbot. We construct the performance model by applying these overheads to the model. We then verify this model by comparing the predicted results with the experimental results from the seven-agent parallel crawler prototype. Using the model, we show that FC2L-UDCbot can crawl Google-scale Web pages within several days using dozens of agents.

The rest of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we present the detailed architecture and implementation of FC2L-UDC. In Section 4, we evaluate the performance of the parallel crawler using FC2L-UDC. In Section 5, we construct a cost model for estimating the crawling speed of the crawler. In Section 6, we conclude the paper.

2. RELATED WORK

In this section, we introduce the key aspects of Web crawlers and the methods for improving their efficiency. Especially, we focus on URL duplication checking for crawling efficiency. In Section 2.1, we first describe the basics of a Web crawling system and review the techniques for parallel Web crawlers. In Section 2.2, we review the techniques for URL duplication checking.

2.1 Web Crawlers

Web crawlers download Web pages from WWW by recursively fetching links (i.e., URLs) from a set of seed Web pages [4, 11]. The primary purpose of Web crawlers is acquisition of a large collection of Web pages to be indexed by the search engine. Fig. 1 shows the Web crawler’s basic modules and their operations.

Parallelization of a Web crawler is an important technique for increasing efficiency of the Web crawler [5, 12, 13]. With the enormous growth of the size of World Wide
Web, parallelization of the Web crawler becomes imperative in order to finish downloading pages in a reasonable amount of time [5]. There are two possible alternatives for the parallel architecture: centralized or decentralized [5, 12].

**Centralized Architecture:** Centralized parallel crawlers [6] follow a client-server model. In such an architecture, a coordinator (i.e., central server) manages the entire crawling process determining the URLs to crawl and controlling all the agents (i.e., clients). A well-known example is the Google crawler [6]. A centralized parallel crawler has the advantage that it ensures fault-tolerance and load balancing through easy manageability and control since the central coordinator has all the global status information. However, it has the disadvantage that the coordinator can be the bottleneck requiring a careful design of the modules.

**Decentralized Architecture:** Decentralized parallel crawlers [5, 7, 12] do not have any central server to manage the crawling. In such an architecture, multiple agents autonomously coordinate all the Web crawling tasks and handle the crawling process independently. A decentralized parallel crawler has the advantage that it provides high speed by allowing the agents to be close to the locations of the Web servers from which they download the Web pages. However, it has the disadvantage of having to address many design challenges and of difficulty in controlling and managing due to unavailability of global status information.

### 2.2 URL Duplication Checking

URL duplication checking is the process of comparing given URLs with the list of URLs already crawled to find new URLs. It is known to be a significant bottleneck for large-scale Web crawlers since the enormous size of the World Wide Web implies a huge number of URLs to compare [4, 10, 11]. We review two techniques proposed by Heydon *et al.* [11] and Lee *et al.* [10].

Heydon *et al.* [11] propose a scalable URL duplication checking technique by using both disk and memory. As the amount of crawled URLs increases, the memory data structures fill up. Therefore, disk usage becomes essential for scalability. In this technique, all the URLs that have already been crawled are stored in disk, and popular URLs and recently accessed URLs are kept in main memory. Lee *et al.* [10] have proposed an extremely efficient technique called Disk Repository with Update Management (DRUM), which extends Heydon’s technique. Here, all the URLs that have already been crawled are stored in disk in a sorted manner and the URLs are checked for duplication in a deferred, batch mode.

### 3. FULL-COVERAGE TWO-LEVEL URL DUPLICATION CHECKING

In this section, we propose a full-coverage two-level URL duplication checking method (full-coverage-2L-UDC) that achieves high-speed URL duplication checking for a parallel Web crawler while ensuring maximum coverage. In Section 3.1, we propose 2L-UDC that reduces the communication between the coordinator and the agents. For efficient URL duplication checking, we adopt DRUM [10] in various parts of the parallel
crawler. DRUM is the first work that shows the crawling speed of the crawler is dependent on the network cost rather than URL duplication checking by showing that URL duplication checking is faster than network bandwidth. However, there have been no research efforts to show such claim in a parallel environment. We show that the claim in DRUM is still valid in a parallel environment as well. In Section 4.3.2, we show this by the actual measurement using a real crawler. We also explain the coverage problem that is incurred by 2L-UDC. In Section 3.2, we propose full-coverage-2L-UDC that solves the coverage problem. In Section 3.3, we present the architecture of the centralized parallel crawler employing full-coverage-2L-UDC. In Section 3.4, we present the implementation details.

3.1 Two-Level URL Duplication Checking (2L-UDC)

The cost of communication between the coordinator and the agents poses an important issue in a centralized architecture for parallel crawling. This communication takes place when the coordinator distributes the URLs to crawl to the agents and when the agents send back the newly discovered URLs to the coordinator. There are two different levels of communication between the coordinator and the agents: the Web page level and the Web site level. We introduce 2L-UDC that communicates at the Web site level in a centralized parallel crawler. 2L-UDC performs site-level URL duplication checking (simply, site-level UDC) in the coordinator while performing page-level URL duplication checking (simply, page-level UDC) in the agents. We adapt DRUM to the Web sites in the coordinator and Web pages in the agents for efficient URL duplication checking. The coordinator obtains unique site URLs through site-level UDC, partitions them, and distributes the partitioned site URLs to the agents. The agents crawl the Web pages of each site in the assigned set of site URLs while performing page-level UDC and send back the newly discovered site URLs to the coordinator. 2L-UDC can greatly reduce the communication costs by communicating at the Web site level instead of the Web page level because the number of Web sites is much smaller than the number of Web pages.

Nevertheless, we observe that 2L-UDC has the so-called coverage problem that has been addressed by Cho and Garcia-Molina [5]. Coverage refers to the percentage of the Web pages discovered or downloaded by the crawler [5, 13]. It is an important metric with respect to a search engine as it has a significant effect on the recall of the search engine. In this paper, the coverage problem means that some Web pages might not be crawled even though the Web pages are discovered by the crawler. This stems from the fact that the Web sites to crawl are assigned to the agents in 2L-UDC, and consequently, the agents crawl only Web pages in the assigned Web sites. Fig. 2 shows an example of the coverage problem. We suppose (1) the whole World Wide Web consists of sites $S_1$ and $S_2$ only, (2) there are two agents $A_1$ and $A_2$, and (3) $S_1$ is assigned to $A_1$ and $S_2$ to $A_2$. Crawling begins from the root page $a$ and $f$ of the corresponding site. While the agent $A_1$ crawls the site $S_1$, only pages $a$, $b$, $c$, and $d$ are crawled because only these pages are directly accessible from the root page of site $S_1$. When the link from page $b$ to page $i$ is discovered, $i$ is not crawled because it is not a page of site $S_1$. Similarly, while the agent $A_2$ crawls the site $S_2$, pages $f$, $g$, $h$, and $j$ are crawled, but page $e$ of site $S_1$ is not crawled even though it has been discovered.
3.2 Full-Coverage-2L-UDC

In this paper, we propose Full-Coverage-2L-UDC that is the first solution for the coverage problem in the centralized architecture. Cho et al. [5] have proposed a solution for the coverage problem in the decentralized architecture. In the decentralized architecture, every agent needs to know which sites to crawl are assigned to which agents since there is no coordinator. For simplicity, Cho et al. [5] have assumed that sites to crawl are known apriori and are partitioned to the agents before crawling. According to this assumption, each agent sends the page URLs with the coverage problem to the corresponding agents.

Unlike in the decentralized architecture, when discovering new sites from the crawled pages, we dynamically partition and distribute them to the agents. For this, Full-Coverage-2L-UDC makes the coordinator keep track of the page URLs that do not belong to the current site (simply, inter-site page URLs) for each agent. The agents exchange these inter-site page URLs with one another. To do that, the coordinator keeps the set of site URLs partitioned by the agents (simply, site partition information) and broadcasts it to the agents for newly discovered sites.

During the process of exchanging the inter-site page URLs among the agents, three different cases may exist; (1) A page URL may belong to a site that is assigned to the current agent. In this case, the agent downloads the Web page corresponding to the page URL; (2) A page URL may belong to a site that is assigned to a different agent. In this case, the current agent dispatches the page URL to the responsible agent; (3) A page URL may belong to a site that has not yet been assigned to any agent. In this case, the current agent sends the site URL of this page to the coordinator as a new site discovered, and then, dispatches the page URL to the responsible agent after the site of the page is assigned to an agent by the coordinator.

3.3 Centralized Parallel Crawler Using Full-Coverage-2L-UDC

We propose an architecture for the centralized parallel crawler using full-coverage-2L-UDC, which we call FC2L-UDCbot. Fig. 3 shows the proposed architecture of FC2L-UDCbot. It consists of one coordinator and multiple agents. The coordinator is the
central manager that controls the entire crawling process. It dispatches site URLs to crawl to the agents and receives new site URLs from the agents. The agents perform the actual Web crawling and send the newly discovered site URLs to the coordinator.

![Diagram of the architecture of FC2L-UDCbot]

(a) The overall architecture of FC2L-UDCbot.

(b) The architecture of the coordinator.

(c) The architecture of an agent.

The modules in the coordinator are the DNS resolver module and the site-level UDC module. The DNS resolver module resolves the hostname of each URL to IP to check if the URL is available. For the DNS resolver, we use the standard solution [14]. The site-level UDC module finds unique site URLs by performing duplication checking for the site URLs. For this, we use a DRUM, SiteURL DRUM, which takes site URLs as the input and returns unique site URLs as the output. The site-level UDC module stores the unique site URLs in Site URL List and distributes them to the agents.
The modules in the agents are the Web page downloader module, link extractor module, robots check module, page-level UDC module, and site partition information module. First, the Web page downloader module performs the actual Web crawling for the pages of the sites given by the coordinator. Second, the link extractor module extracts page URLs from each downloaded Web page and stores them in Page URL List. Third, the robots check module verifies if the page URLs in Page URL List are allowed to crawl. Fourth, the page-level UDC module performs duplication checking for the page URLs that are passed from the robots check module. For this, we use a DRUM, PageURL DRUM, which gets page URLs as the input and returns unique page URLs as the output. Fifth, the site partition information module dispatches the unique page URLs to the responsible agents to solve the coverage problem. To find the responsible agent of each page, we use SitePartitionInformation DRUM, which has two operations: (1) storing site partition information given by the coordinator and (2) finding the responsible agent for the site of each page from the site partition information. For the former, the key is the hash value of the site, and the value is the responsible agent for the site. Such <key, value> tuples are stored in Cache Z of SitePartitionInformation DRUM. For the latter, the responsible agent for the site is found from Cache Z using the hash value of the site as the key.

3.4 Implementation

In this section, we explain the implementation details for FC2L-UDCbot. We separate the overall algorithm into two parts: one for the coordinator and the other for an agent. We call them FC2L-UDCbot_Coordinator and FC2L-UDCbot_Agent, respectively.

3.4.1 Algorithm for the coordinator of FC2L-UDCbot

Fig. 4 shows the algorithm FC2L-UDCbot_Coordinator(). The inputs are SeedSiteURLsList, the list of seed site URLs, and TargetNumOfWebPages, the target number of Web pages to be crawled. The output is DownloadedWebPages, the set of downloaded Web pages. In Step 1, variables are initialized. In Step 2, crawling is done by communicating with agents. Steps 2.1-2.2 are performed repeatedly until TargetNumOfWebPages is crawled. In Step 2.1, site-level URL duplication checking is done using SiteURL DRUM, and SitePartitionInfo is constructed by assigning all unique URLs to each agent i using a round-robin scheme. In Step 2.2, SitePartitionInfo is sent to each agent i by executing the algorithm FC2L-UDCbot_Agent() in Fig. 5. Then, newly discovered site URLs and the number of downloaded Web pages are received from each agent i as the output of the algorithm FC2L-UDCbot_Agent().

Algorithm FC2L-UDCbot_Coordinator();
Input: (1) SeedSiteURLsList /* Seed site URLs */
       (2) TargetNumOfWebPages /* No. of pages to be downloaded */
Output: (1) DownloadedWebPages /* Set of downloaded Web pages */
Algorithm:
/*Step 1. Initialize variables */
NumOfDownloadedWebPages := 0; /* No. of downloaded Web pages */
SiteURLsToCrawlList := SeedSiteURLsList; /* List of site URLs to crawl is initialized as SeedSiteURLsList */

/* Step 2. Crawl the Web pages by communicating with agents */
WHILE NumOfDownloadedWebPages<TargetNumOfWebPages DO
BEGIN
  /* Step 2.1. Perform site-level UDC for SiteURLsToCrawlList and construct SitePartitionInfo */
  UniqueSiteURLsList := Ø; /* List of unique site URLs to crawl */
  SitePartitionInfo := Ø; /* Site URLs partitioned by agents */
  FOR each site in SiteURLsToCrawlList DO
    /* site-level UDC using SiteURL DRUM */
    UniqueSiteURLsList.append(SiteURLDRUM.findUniqueURLs(site));
    Construct SitePartitionInfo by assigning site URLs in UniqueSiteURLsList to the agents in a round-robin fashion;
  END
  /* Step 2.2. Send SitePartitionInfo to each agent and receive newly discovered site URLs */
  SiteURLsToCrawlList := Ø;
  FOR i=0 to NumOfAgents-1 DO
    /* Step 2.2.1. Send SitePartitionInfo to each agent */
    <SiteURLsToCrawlList[i], NumOfDownloadedWebPages[i]> := FC2L-UDCbot_Agent(SitePartitionInfo);
    /* Step 2.2.2. Receive newly discovered site URLs from each agent and store them into SiteURLsToCrawlList */
    SiteURLsToCrawlList.append(SiteURLsToCrawlList[i]);
    NumOfDownloadedWebPages := NumOfDownloadedWebPages + NumOfDownloadedWebPages[i];
  END
END

Fig. 4. The algorithm for the coordinator of FC2L-UDCbot.

3.4.2 Algorithm for the agent of FC2L-UDCbot

Fig. 5 shows the algorithm FC2L-UDCbot_Agent(). The input is SitePartitionInfo. The output consists of NewSiteURLsList, the list of newly discovered site URLs, and NumOfDownloadedWebPages, the number of newly downloaded Web pages by the agent. The algorithm consists of the following five steps. In Step 1, variables are initialized. In Step 2, SitePartitionInfo is stored into SitePartitionInformation DRUM by executing the procedure Agent_StoreSitePartitionInfo(). In each agent, we maintain two global data structures that are used for resolving the coverage problem: AssignedPageURLsList and UnassignedPageURLsList. AssignedPageURLsList is the list of page URLs that are assigned to an agent among those in PageURLsWithCoverageProblemList, the list of page URLs with the coverage problem, and UnassignedPageURLsList is the list of page URLs that are not assigned to any agent among those in PageURLsWithCoverageProblemList. They are initialized at the beginning of Web crawling by each agent. In Fig. 6, the procedure Agent_StoreSitePartitionInfo() stores SitePartitionInfo into SitePartitionInformation DRUM in Step 1; it moves the entries of the page URLs corresponding to each site URL in SitePartitionInfo from UnassignedPageURLsList to AssignedPageURLsList in Step 2.
In Step 3 of Fig. 5, pages are crawled from each site by using breadth-first search. First, the sites assigned to the current agent are identified from SitePartitionInfo.

Algorithm FC2L-UDCbot_Agent();
Input: (1) SitePartitionInfo /* Site URLs partitioned by agents */
Output: (1) NewSiteURLsList /* List of site RLs newly discovered */
        (2) NumOfDownloadedWebPages /* Total number of Web pages newly ownloaded */

Algorithm:
/* Step 1. Initialize variables */
NewSiteURLsList : = Ø ; NumOfDownloadedWebPages := 0;
PageURLsWithCoverageProblemList := Ø; /* List of page URLs with the coverage problem */
Agent_StoreSitePartitionInfo(SitePartitionInfo);
/* Step 2. Store SitePartitionInfo to SitePartitionInformation DRUM */
Agent_StoreSitePartitionInfo(SitePartitionInfo);
/* Step 3. Crawl pages by site until termination conditions are satisfied */
Find the sites assigned to the current agent in SitePartitionInfo and store them into AssignedSiteURLsList ;
FOR each site in AssignedSiteURLsList DO
BEGIN
PageURLsToCrawlList := Ø; /* List of page URLs to crawl */
UniquePageURLsList := Ø; /* List of unique page URLs discovered during crawling */
PageURLsToCrawlList.append(rootpage(site));
/* Termination conditions */
WHILE ((ThresholdPagesPerSite or ThresholdDepthPerSite are not reached) OR PageURLsToCrawlList is not empty) DO
BEGIN
/* Step 3.1. Crawl each page and extract new page URLs from the page */
page := PageURLsToCrawlList.remove();
Download page; NumOfDownloadedWebPages += 1;
Extract URLs from page and store them into PageURLsList;
/* Step 3.2. Perform page-level UDC for the extracted page URLs */
UniquePageURLsList.append(PageURLDRUM.findUniqueURLs(PageURLsList));
/* Step 3.3. Add pages without coverage problem into PageURLsToCrawlList; add those with coverage problem into PageURLsWithCoverageProblemList */
FOR each page in UniquePageURLsList DO
BEGIN
IF site (page) is the current site THEN
PageURLsToCrawlList.append (page);
ELSE
BEGIN
PageURLsWithCoverageProblemList.append(page);
/* Find newly discovered sites using SitePartitionInformationDRUM and store them into NewSiteURLsList */
NewSiteURLsList.append(
SitePartitionInformationDRUM.findUniqueSiteURLs(site(page)))
END
END
/* Step 4. Solve the coverage problem for PageURLsWithCoverageProblemList */
Agent_SolveCoverageProblem(PageURLsWithCoverageProblemList);

/* Step 5. Return NewSiteURLsList and NumOfDownloadedWebPages */
RETURN <NewSiteURLsList, NumOfDownloadedWebPages>;

Fig. 5. The algorithm for the agent of FC2L-UDCbot.

Then, the root page of each site is added to PageURLsToCrawlList, which is the list of page URLs to crawl for each site. The following three steps are repeated for each page URL in PageURLsToCrawlList until one of the following conditions is met: (1) ThresholdPagesPerSite, the threshold number of pages per site, has been downloaded; (2) ThresholdDepthPerSite, the threshold depth per site, is reached; or (3) PageURLsToCrawlList becomes empty. The three steps are: (1) downloading the page URL and extracting URLs from the downloaded Web pages; (2) performing page-level UDC for the extracted URLs and (3) adding URLs to PageURLsToCrawlList if the site of each page URL is the current crawling site; otherwise, adding them to PageURLsWithCoverageProblemList and finding newly discovered site URLs using SitePartitionInformation DRUM.

In Step 4, the coverage problem is resolved by executing the procedure Agent_SolveCoverageProblem() in Fig. 6. The input to the procedure Agent_SolveCoverageProblem() is PageURLsWithCoverageProblemList. In Step 1, the page URLs in PageURLsWithCoverageProblemList are stored into AssignedPageURLsList or UnAssignedPageURLsList according to the responsible agent for each page obtained from SitePartitionInformation DRUM. If the responsible agent is the current agent, the page URL is sent to the Web page downloader of that agent. If the responsible agent is a different agent, the pair of page URL and the responsible agent is added to AssignedPageURLsList. If the page URL is not assigned to any agent, the page URL is added to UnassignedPageURLsList. In Step 2, the page URLs in AssignedPageURLsList are dispatched to the responsible agents: (1) the page URLs in AssignedPageURLsList are sent to the responsible agents; (2) the page URLs assigned to the current agent from other agents are received.

In Step 5, the algorithm FC2L-UDCbot_Agent() returns NewSiteURLsList and NumOfDownloadedWebPages as the output to the coordinator.

Procedure Agent_StoreSitePartitionInfo();
Input: SitePartitionInfo /*Site URLs partitioned by agents*/  
/* UnAssignedPageURLsList stores page URLs that are not assigned to an agent */  
/* AssignedPageURLsList stores page URLs that are assigned to an agent */
FOR each SiteURL in SitePartitionInfo DO
  BEGIN
    /* Step1. Add each site in SitePartitionInfo to SiteURLPartitionInformationDRUM */
    SitePartitionInformationDRUM.add(<SiteURL, ResponsibleAgent(SiteURL)>);
  END
END
END
/* Step2. Move pages corresponding to the site in U */
AssignedPageURLsList to AssignedPageURLList */
Remove entries for page URLs corresponding to SiteURL from UnAssignedPageURLsList;
Add entries for the page URLs to AssignedPageURLList;
END

Procedure Agent_SolveCoverageProblem();
Input: PageURLsWithCoverageProblemList /* The list of page URLswith the coverage problem*/
/* Step1. Store URLs in PageURLs WithCoverageProblemList into AssignedPageURLsList or UnAssignedPageURLsList */
FOR each PageURL in PageURLsWithCoverageProblemList DO
BEGIN
IF site(PageURL) is in SitePartitionInformationDRUM THEN
/*Case 1: If PageURL is assigned to the current agent*/
IF ResponsibleAgent is the current agent THEN
Send PageURL to Web page downloader of the current agent;
/*Case 2: If PageURL is assigned to a different agent*/
ELSE
AssignedPageURLsList.add(<PageURL, ResponsibleAgent>);
/*Case 3: If PageURL has not yet been assigned to any agent*/
ELSE
UnAssignedPageURLsList.add(PageURL);
END
END

/* Step 2. Dispatch page URLs in AssignedPageURLsList to the responsible agents */
Send page URLs in AssignedPageURLsList to the responsible agents;
Receive PageURLs assigned to the current agent from other agents;

Fig. 6. The procedures for the agent of FC2L-UDCbot.

4. PERFORMANCE EVALUATION

4.1 Emulating High-Speed Network Connection to WWW

We propose an emulated experimental environment to guarantee a high and consistent network bandwidth. To evaluate large-scale Web crawling, a high network bandwidth is required [5, 10]. However, this is generally a constraint in an academic setting [4]. The proposed experimental methodology creates an artificial World Wide Web on disk, which we call DiskWWW. This is done by dumping the crawled pages that were obtained from an actual Web crawl to disk in the order these pages were crawled. Here, we use 8 million Web pages where each page consists of 22.3KB on the average. We use a separate disk for one DiskWWW for each agent in order to have each agent independently connected to the artificial World Wide Web. We use one disk arm for DiskWWW and one process for reading pages from DiskWWW to simulate an Internet con-
nection using one LAN card.

We obtain the emulated target bandwidth by controlling sequential or random reads of disk since the read speed of a Web page from DiskWWW may vary depending on sequential or random reads. Our target network bandwidth is the same as the average Internet (i.e., 320Mbits/sec) reported by Lee et al. [10] in their experiments. Since the data transfer rate of the disks we used is 640Mbits/sec (i.e., the maximum possible crawling speed = 3,587.44 pages/sec), we adjust the read speed for each Web page by suspending to achieve the target bandwidth. Specifically, if the time spent for reading a Web page is greater than the time required for reading it using the target network bandwidth, we add the difference between them to the excess time. When it is less, the difference is subtracted from the excess time. If the excess time is not sufficient, we suspend the crawler activity until the time required for reading the page with the target network bandwidth is reached.

4.2 Experimental Environment

The purpose of our experiments is to show that how the overheads in a parallel environment are effectively handled by our URL duplication checking method. First, we show the crawling speeds of the parallel crawler (1) using full-coverage-2L-UDC and (2) using 2L-UDC to evaluate the overhead of resolving the coverage problem. We use the crawling speed defined in Eq. (1) as the measure. Second, we show the crawling speed of the parallel crawler using full-coverage-2L-UDC and that of the single-machine crawler using the original DRUM [10]. There have been no published results for a centralized parallel Web crawler since the only effort is the crawlers of commercial search engines such as Google. We only compare with DRUM, which uses the most efficient URL duplication checking method on a single machine. In Section 4.3.2, we compare the crawling speed of the parallel crawler using full-coverage-2L-UDC with that of DRUM. We use n to denote the number of agents and m to denote the number of Web pages in the unit of one million. We use five different n’s for the parallel configurations: one, two, three, five, and seven. Thus, each agent communicates with zero, one, two, four, and six agents (i.e., n – 1), respectively. We use four different m’s: one, two, four, and eight. We perform all the experiments using the same seed URLs. The seed URLs are collected from the directory service of Open Directory Project [15], which provides the URLs of trust-worthy pages.

\[
\text{CrawlingSpeed} = \frac{\text{The Total Number of Pages Crawled}}{\text{The Total Crawling Time}}.
\]  

(1)

We use eight Linux PCs, each with Core2Duo E6750, 1GB of main memory, and a 400GB Seagate SATA2 disk. All PCs have the same configuration. One of them is the coordinator, and the others are agents. Our environment is the same as utilizing two LAN cards where one is the emulated external connection to DiskWWW with the speed of 320 Mbits/sec (i.e., 40 MBytes/sec), and the other is the real internal connection between the coordinator and the agents with the speed of 100 Mbits/sec (i.e., 12.5 MBytes/sec). That is, we crawl Web pages of size 22.3KB on the average from DiskWWW at 320 Mbits/sec.
4.3 Results of the Experiments

4.3.1 Overhead of resolving the coverage problem

Fig. 7 shows the crawling speed of the parallel crawler using full-coverage-2L-UDC and using 2L-UDC as the number of agents varies. The result clearly indicates that the overhead of full-coverage-2L-UDC is not significant. Specifically, the crawling speed of the parallel crawler with seven agents using full-coverage-2L-UDC is 12,252 pages/sec indicating only 1.2% decrease in the speed compared with 12,399 pages/sec when using 2L-UDC. This insignificance of the overhead stems from the fact that the Web pages have links to the Web pages largely in the same site rather than to those in the other sites [16].

4.3.2 Comparison with DRUM

Fig. 8 shows the crawling speed of the parallel crawler using full-coverage-2L-UDC and that of the single-machine crawler using the original DRUM as for the parallel crawler varies. We observe that the crawling speed of FC2L-UDCbot is improved by 1.9 to 6.9 times as compared with that of the single-machine crawler. The result shows that the crawling speed of FC2L-UDCbot with one agent is slightly lower than that of the single-machine crawler due to the communication overhead incurred between the coordinator and the agent. The overhead is not significant being only 0.58%. Here, we set the target crawling speed of the single-machine crawler using the original DRUM in our emulated network environment to be as close to that of the real network environment used by Lee et al. [10] as possible; it turned out to be 1,784.22 pages/sec\(^1\). This result shows the validity of our experiments in our emulated network environment. The result also shows that the crawling speed of FC2L-UDCbot is approximately proportional to the number of agents. Specifically, the parallel crawler with seven agents outperforms it with one agent by 6.91 times. This result indicates that full-coverage-2L-UDC allows the crawling speed of the parallel crawler to scale well through parallelization of the DRUM approach. We obtain 12,251.75 pages/sec as the crawling speed of FC2L-UDCbot with seven agents. Hence, it can crawl 30 billion Web pages (equivalent to what Google does) in 29 days in our experimental setting.

---
\(^1\) Naturally, it was made almost equal to 1,789 pages/sec reported by Lee et al. [10] since the network bandwidths are 320Mbits/sec in both cases, and the crawling speeds are both network bound.
5. COST ANALYSIS

In Section 5.1, we analyze the overheads that are incurred in the centralized parallel crawler. In Section 5.2, we construct a cost model for estimating the crawling speed of the centralized parallel crawler. In Section 5.3, we verify the cost model by comparing the estimated and experimental crawling speeds. In Section 5.4, based on the model, we find the maximum number of agents that can be used for the centralized parallel crawler in a network with a given bandwidth.

5.1 Analysis of Overheads

The overheads that are incurred in the parallel configuration for the centralized parallel crawler are affected by two parameters, namely, the number of agents \( n \) and the number of Web pages \( m \). The greater the number of agents is, the greater the cost of communication between the coordinator and the agents and between the agents themselves is. Similarly, the greater the number of Web pages to crawl, the greater the number of URLs that are communicated between the agents (and the coordinator).

Fig. 9 shows the overhead for an agent in the centralized parallel crawler in terms of the crawling speed as \( n \) varies when \( m = 1 \) (i.e., 1 million Web pages). Fig. 10 shows the overhead for an agent in FC2L-UDCbot in terms of the crawling speed as \( n \) varies when \( m = 7 \). We analyze the types of overhead incurred in an agent in FC2L-UDCbot compared with that in the single-machine crawler. There are four main activities incurring the overhead in the parallel configuration. The overheads for each agent are (1) inter-agent setup overhead; (2) coordinator-agent setup overhead; (3) site URL sending overhead; (4) site partition information overhead; and (5) page URL exchange overhead. The inter-agent setup overhead is the overhead incurred by the initial setup of the connection between the agents. The coordinator-agent setup overhead is the overhead incurred by the initial setup of the connection between the coordinator and the agents. The site partition information overhead is the overhead of receiving and storing the site partition in-
formation from the coordinator. The page URL exchange overhead is the overhead of exchanging page URLs between the agents to resolve the coverage problem.

We now analyze each overhead from the results in Figs. 9 and 10. The inter-agent setup overhead and the coordinator-agent setup overhead are not affected by both $n$ or $m$ since they are incurred once for each agent. In principle, this cost is expected to increase as increases. However, the marginal increase is measured to be negligible and is not reflected here. We collectively refer to these as FixedOverhead. Figs. 9 and 10 show that FixedOverhead is 4.04.

---

2 It increases from 0.446 pages/sec to 0.450 pages/sec when varies from one to seven. With the formula of $0.446 + 0.004 \times \frac{(n-1)}{6}$, the inter-agent setup overhead in a 200-node configuration is expected to be only 0.579 pages/sec.
We estimate the site URL sending overhead, the site partition information overhead, and page URL exchange overhead from the results in Figs. 9 and 10. The site URL sending overhead decreases as the number of agents increases. This trend stems from the fact that, given a fixed \( m \) (here, \( m = 1 \)), the number of the site URLs discovered by an agent decreases as the number of agents increases. We use the maximum value (i.e., 1.11) of overheads measured at consecutive points as an initial overhead. On the other hand, the site URL sending overhead increases linearly as the number of Web pages to be crawled increases exponentially since the number of Web sites increases approximately linearly as the number of Web pages increases exponentially [17, 18]. The number of pages for each site follows a power law distribution [17]. We use the maximum difference (i.e., 0.36) of overheads measured at consecutive points as the coefficient of the logarithmic function. Thus, we use \( 1.11+0.36\cdot \log_2 m \) as the formula to estimate the Site URL Sending Overhead.

The site partition information overhead increases linearly as the number of agents increases. Here, the amount of site partition information (i.e., the number of the pair \(<\text{site}, \text{agent}>\)) to send to each agent is constant regardless of the number of agents. However, the reason why the site partition information increases as the number of agents increases is that we transfer only once agent information on the sites assigned to the same agent. We use the maximum difference (i.e., 0.28) of the overheads measured at consecutive points as the coefficient of the linear function. On the other hand, the site partition information overhead increases linearly as the number of Web pages to be crawled increases exponentially since the amount of site partition information linearly depends on the number of Web sites, and the number of Web sites increases approximately linearly as the number of Web pages increases exponentially [17, 18]. We use the maximum difference (i.e., 1.59) of overheads measured at consecutive points as the coefficient of the logarithmic function. Thus, we use \( 3.83+0.28n+1.59\cdot \log_2 m \) as the formula to estimate the site partition information overhead.

The page URL exchange overhead increases as the number of agents to communicate with (i.e., \( n-1 \)) increases. We use the maximum difference (i.e., 4.39) of overheads measured at consecutive points as the coefficient of the linear function. On the other hand, the page URL exchange overhead increases linearly as the number of Web pages to be crawled increases exponentially. The number of page URLs exchanged depends approximately linearly on the number of sites since the number of page URLs exchanged per site stays relatively constant\(^3\). The possible reason for this is that the number of page URLs exchanged per site is not necessary proportional to the total number of Web pages in the site, but instead is mainly affected by the other factors such as the quality of the site\(^4\). We use the maximum difference (i.e., 5.5) of overheads measured at consecutive points as the coefficient of the logarithmic function. Thus, we use \( 4.39 \cdot (n-1) + 5.5 \cdot \log_2 m \) as the formula to estimate the page URL exchange overhead.

### 5.2 Cost Model for the Crawling Speed

In this section, we construct a cost model for estimating the crawling speed of FC2L-UDCbot. Eq. (2) shows each component overhead composing TotalOverheadFor-
AnAgent; Eq. (3) shows the estimated crawling speed for an agent taking the overheads into account. CrawlingSpeedOfSingleMachine is the speed of the single machine crawler. Here, we use 1784.22 pages/sec, which is measured in Section 5.2.2, as CrawlingSpeedOfSingleMachine. We note that, in the typical large-scale configuration such as \( n = 200 \) and \( m = 30,000 \), the overhead is dominated by \( n \).

\[
\text{TotalOverheadForAnAgent} = \text{SiteURLSendingOverhead} + \text{SitePartitionInformationOverhead} + \text{PageURLExchangeOverhead} + \text{FixedOverhead} = (1.11 + 0.36 \log_2 m) + (3.83 + 0.28 n + 1.59 \log_2 m) + (4.39 (n - 1) + 5.5 \log_2 m) + (4.04) = 4.59 + 4.67 n + 7.45 \log_2 m \text{(pages/sec)}
\]

(2)

\[
\text{CrawlingSpeedForAnAgent} = \text{CrawlingSpeedOfSingleMachine} - \text{TotalOverheadForAnAgent} = 1784.22 - (4.59 + 4.67 n + 7.45 \log_2 m) \text{ (pages/sec)}
\]

(3)

5.3 Verifying the Cost Model

The overall speed of FC2L-UDCbot is the number of agents times the crawling speed per agent. Fig. 11 (a) shows a comparison between the estimated and experimental crawling speeds as \( n \) varies when \( m \) is 1.0; Fig. 11 (b) shows the same as \( m \) varies when \( n \) is seven. The results indicate that our equations produce quite accurate estimation.

5.4 The Maximum Number of Agents for FC2L-UDCbot

Lee et al. [10] have demonstrated that the performance of DRUM is dependent on the network cost rather than URL duplication checking since URL duplication checking in DRUM is more efficient than the average network bandwidth. Lee et al. [10] show that DRUM would ideally yield an average download speed of 4,192 pages/sec by analyzing the disk overheads of DRUM, which is faster than the average network bandwidth in their experimental environment (i.e., 320 MBits/sec=1,789 pages/sec) while Mercator [11] would yield an average download speed of 1.4 pages/sec and Polybot [4] 0.2 pages/sec. Lee et al. [10] argue that, when \( S \) is the crawling speed of one agent and there are \( n \) agents, the crawling speed of \( n S \) is possible assuming the network bandwidth is \( n \) times as large as the average network bandwidth in their experiments. However, in practice, there exists limited network bandwidth, and hence, there is a limit on the maximum number of agents.

Now, we find the maximum number of agents of FC2L-UDCbot that can be supported simultaneously under the limited network bandwidth. One agent in our experimental environment consumes 320 Mbits/sec for crawling. In the real Web environment with a current maximum Internet speed of 10Gbits/sec [10], FC2L-UDCbot can support around 31 agents simultaneously when every agent shares the Internet. Eq. (4) shows the time duration for crawling 30 billion Web pages using FC2L-UDCbot with the parallel configuration. According to the equation, FC2L-UDCbot with 31 agents can crawl 30 billion Web pages in 7.4 days.

---

5 They did not take into account site partition information overhead and page URL exchange overhead since they did not consider 2L-UDC.
A FULL-COVERAGE TWO-LEVEL URL DUPLICATION CHECKING ...

\[(1,000,000m)\text{(pages)}\]
\[
\frac{\text{(Crawling Speed For An Agent) } n\text{(pages/sec)}}{(1,000,000 \times 30,000)\text{(pages)}} =
\]
\[
\frac{(1784.22 - (4.59 + 4.67 \times 31 + 7.45 \cdot \log(30,000))) \times 31\text{(pages/sec)}}{(1,000,000 \times 30,000)\text{(pages)}} = 634,976.27\text{ seconds} = 7.4\text{ days.}
\]

\[m = \frac{4,976.27\text{ seconds}}{7.4\text{ days}} = \frac{4,976.27\text{ seconds}}{7 \times 24 \times 60 \times 60\text{ seconds}} = 0.076\text{ pages/second.}
\]

Fig. 11. Comparison between experimental and estimated crawling speeds.

(a) Varying \(n\).

(b) Varying \(m\).

Fig. 12. The crawling speed of FC2L-UDCbot as \(n\) increases.
In a future Internet with a maximum projected speed of 100Gbits/sec [19], FC2L-UDCbot will be able to support 313 agents simultaneously. Fig. 12 shows the crawling speed of FC2L-UDCbot as \( n \) increases when \( m = 30,000 \) by using the denominator of Eq. (4). It indicates that FC2L-UDCbot with a 180-agent configuration has the highest performance. This result shows that the overhead increases as \( n \) increases, and consequently, the overhead is higher than the increased crawling speed as the agents are added from a 180-agent configuration. With 180 agents, FC2L-UDCbot should be able to collect 30 billion Web pages in 55.8 hours.

6. CONCLUSIONS

In this paper, we have proposed full-coverage-2L-UDC, which is an efficient URL duplication checking method ensuring maximum coverage. We have introduced 2L-UDC for a centralized parallel crawler to reduce the communication cost between the coordinator and the agents. Furthermore, having observed that the coverage problem occurs in 2L-UDC, which reduces recall of the search engine, we have proposed a method to resolve the coverage problem. We have proposed FC2L-UDCbot, which is an architecture of the centralized parallel crawler using full-coverage-2L-UDC. We have built a seven-agent FC2L-UDCbot for extensive experiments. We have shown that the crawling speed of FC2L-UDCbot is approximately proportional to the number of agents (i.e., FC2L-UDCbot is faster than a single-machine crawler by 6.9 times). Full-coverage-2L-UDC allows FC2L-UDCbot to be scalable to the number of agents since it effectively deals with the overheads incurred in a parallel environment.

We have constructed a performance model to show the efficiency and scalability of FC2L-UDCbot. For this, we have proposed an emulated Web crawling environment in order to overcome the network bandwidth limitation. Then, we have built a seven-agent parallel crawler for extensive experiments. We have analyzed the overheads incurred in FC2L-UDCbot and have constructed the performance model. We have then verified the model. Using the model, we show that FC2L-UDCbot can crawl Google-scale Web pages within several days using dozens of agents.

Our experimental results and cost model are meaningful in the sense that we project the performance for a large-scale crawler based on the actual measurement using a small-scale crawler. It is very difficult for an academic institute to build a real-scale crawler because of limited resources. Therefore, an elaborate cost model is needed to test and project the performance and the scalability of a large-scale system without actually building it. To achieve this, we have first proposed a new URL duplication checking method that resolves the overheads incurred in a parallel environment effectively. Second, we have measured the actual crawling speed and the overheads incurred in a parallel environment by an extensive experiment using a real crawler. Finally, we have constructed a rigorous cost model based on the overheads obtained from the actual measurement to project the crawling speed for a large-scale crawler.
REFERENCES


**Arjumand Younus** is a joint Ph.D. candidate at National University of Ireland, Galway and University of Milano-Bicocca, Milan. She holds an M.S. in Computer Science from KAIST, South Korea and a B.S. in Computer Science from the University of Karachi. Her primary research interests lie in information retrieval and social media analytics. She is also a recipient of the prestigious Google Anita Borg Award for women in computing within the Europe, Middle East and Africa region.

**Kyu-Young Whang** earned his Ph.D. from Stanford University in 1984. From 1983 to 1991, he was a Research Staff Member at IBM T. J. Watson Research Center. In 1990, he joined KAIST, where he currently is a Distinguished Professor at Department of Computer Science. His research interests encompass database systems/storage systems, search engines, object-oriented databases, geographic information systems, and data mining. He was the general chair of VLDB2006. He served as an Editor-in-Chief of the VLDB Journal from 2003 to 2009. He is a Fellow of ACM and IEEE. He served as the chair of IEEE Technical Committee on Data Engineering (TCDE) from 2003 to 2004.

**Hyuk-Yoon Kwon** received the B.S. in Computer Science and Statistics from University of Seoul (UOS) in 2005, the M.S. in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2007, and the Ph.D. in Computer Science from KAIST in 2013. His research interests include GIS, information retrieval, search engines, and big data analytics.

**Yeo-Mi Yeo** received the B.S. in Computer Science from Korea University in 2006, the M.S. in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2008. Her research interests include information retrieval and search engines.