

Collaborative Web Crawling: Information Gathering/Processing over Internet

Shang-Hua Teng* UIUC steng@cs.uiuc.edu	Qi Lu† IBM qilu@yahoo-inc.com	Matthias Eichstaedt IBM eichstam@almaden.ibm.com
Daniel Ford IBM dford@almaden.ibm.com	Tobin Lehman IBM lehman@almaden.ibm.com	

Abstract

The main objective of the IBM Grand Central Station (GCS) is to gather information of virtually any type of formats (text, data, image, graphics, audio, video) from the cyberspace, to process/index/summarize the information, and to push the right information to the right people. Because of the very large scale of the cyberspace, parallel processing in both crawling/gathering and information processing is indispensable. In this paper, we present a scalable method for collaborative web crawling and information processing. The method includes an automatic cyberspace partitioner which is designed to dynamically balance and re-balance the load among processors. It can be used when all web crawlers are located on a tightly coupled high-performance system as well as when they are scattered in a distributed environment. We have implemented our algorithms in Java.

1 Introduction

The cyberspace is a vast media for rapid information distribution. Internet surfing is now a popular way for people and industries to rapidly gather information. However, because of the immense amount of information available in cyberspace, automatic information gathering, screening, and delivering systems have become necessary.

One such system is the Grand Central Station (GCS) system being developed at the IBM Almaden Research Center. This system combines numerous aspects of information discovery and dissemination into a single, convenient system [4, 6, 2]. GCS performs many functions by providing an infrastructure that supports the discovery and tracking of information in a digital domain such as cyberspace, and disseminates these discoveries to those who have an interest.

*Department of Computer Science, University of Illinois, Urbana, IL 61801. This work is done at IBM Almaden Research Center.

†This work is done at IBM Almaden Research Center.

One of the key components of virtually all information discovery system infrastructures accessing cyberspace (i.e., the Internet) is a Gatherer that systematically crawls the Internet to gather diverse types of data sources and transforms or summarizes them into a single, uniform, meta-data format. This format generally reflects the format found in the system used by the person requesting the information. Webcasting technology referred to as an “internet push” is used to match the summarized information with users’ profiles and re-channel each piece of information to those who need it [2, 3, 4, 7, 6]

This paper describes our design and implementation of a collaborative web crawling system for GCS. Because of the sheer volume of data available, it is necessary to build a web crawling system that uses multiple processors to share the load of information gathering and summarization. Preferably, to make a Gatherer efficient, the system should be scalable.

To build a scalable distributed web crawling system, we need to maximally utilize computing resources, in other words, we need to minimize the overhead induced by the collaboration among processors. To do this, we need to develop an automatic web-space partitioning algorithm for load balancing among crawling processors. Web-space partitioning is a different and much more challenging problem than discussed in current traditional graph partitioning problem studies dealing with VLSI circuit design [1] and parallel scientific computing [8].

For example, one difficulty comes from the fact that a web-space directed graph, used to model the information at the site, is usually not discoverable before the crawling occurs; This is because web sites are dynamic, that is, they are always changing, having information added and deleted up to the point the crawling actually takes place. This constant changing of the information - and therefore the directed graph used to model

the information - prevents directly applying the previously mentioned graph partitioning methods that are designed for static (non-changing) and explicitly given graphs. The lack of full knowledge of a web-graph before a webspace is partitioned also requires the amount of load and the number of hyperlinks across a partition to be changeable at any stage of collaborative crawling, and hence dynamic re-partitioning and load re-balancing are necessary.

Another problem that would need to be overcome is the addressing problem that arises in attempting to partition a webspace. For example, given a uniform resource locator (URL), a quick decision needs to be made as to which partition it would belong. Depending upon the partition, it would then be sent to a designated processor for crawling and processing. Further, because the web-graph is dynamic, a problem can arise in simply organizing a partition.

We present a dynamic partitioning and load-balancing algorithm using one-level graph coarsening. Our algorithm uses the hierarchical structure of URL names to define an intermediate structure called *superpage*, where a superpage is a collection of URLs that share some initial sub-sequence of their URL names. Superpages are used as the basic units for top level partitioning. For example, for a URL such as `cs.cmu.edu/groups/parallel/parallel.html` - a specification of a path in a name tree - may be viewed as a sequence of tokens (`edu`, `cmu`, `cs`, `groups`, `parallel`, `parallel.html`). A superpage whose initial tokens are `cs.cmu.edu/groups/parallel` can be formed, and URLs whose initial tokens form `math.cmu.edu` may also be formed as another superpage. The set of superpages is formed dynamically during the crawling to accommodate the new information. By partitioning and re-partitioning these superpages, our method dynamically balances a processing load incurred in information gathering. At a high level, we coarsen the web-graph using the URL name hierarchy, so that URL-pages from the same superpage will be considered as a single unit.

The coarsening introduces an explicitly structured web-graph where every URL belongs to a superpage. Each superpage contains URL pages that are reasonably local to each other, both in physical addresses and in hyperlink connections. When the web-graph is projected onto the set of superpages that are created dynamically, a smaller "coarsened" image of the original web-graph is obtained where this coarsened image approximates the larger web-graph, both in physical and hyperlink vicinity. A good partition for this coarsened image would most likely be a good partition for the original web-graph.

In our scheme, superpages are automatically recognized and generated. Necessary information is obtained

and maintained to measure the "volumes" of superpages as well as any pattern of connections among superpages. We maintain the access rate for each processor to these superpages to determine if different processors have different access rates to each superpage. For example, a crawler located in IBM-Almaden in the United States may have a longer access time to a server located in Japan than a crawler located in IBM-Tokyo in Japan. Once a set of superpages is formed and processor statistical data is obtained, our method employs partitioning algorithm to automatically generate favored partitions for superpages, and map them among available processors. With these features, our scheme can be used when all web crawlers are located on a tightly coupled high-performance parallel system as well as on a distributed system where processors are scattered.

Our collaborative web crawling system supports a mechanism which allows processors to communicate with each other to coordinate and handle hyperlinks across a partition. This communication mechanism uses the IBM TSpaces [10] - a persistent communication buffer - to support fast and inter-process communication. Logically, a TSpaces can be viewed as a large white-board, or global communication buffer accessible by all processors.

We have implemented the collaborative web crawling scheme in Java as a part of the GCS project. As discussed above, IBM TSpaces is used for the coordination of processors when a dynamic load balancing is needed, and for the communication of hyperlinks across partitions among processors. IBM TSpaces is implemented in Java.

2 Sequential Crawling in GCS: a review

A GCS Gatherer is an expanded web crawler. Based on its configuration, the Gatherer periodically crawls a list of data sources such as web servers, news servers, file directories, and databases. The crawling follows the information organization structure of the corresponding data source such as HTML hyper-links, news group structure, file system hierarchy, and database schemes. For each encountered object such as an HTML page, a news article, a file, or a database table, the Gatherer first retrieves the object from the data source and then creates a summary for it. The summary data is represented using a summary or abstract format such as SML which contains query keywords, thumbnails, and other important information about the data object.

The Gatherer itself has a number of multi-threaded components. It includes a Crawler component that crawls media sources and retrieves objects while a Recognizer component tries to determine the format for each of the retrieved objects. A Summarizer component contains specialized code that enables it to read

a great number of different object formats such as a Freelanx graphics presentation, an HTML page, a Lotus Notes database, or an Excel spreadsheet. It also provides a flexible structure for plugging in customized summarization code to be used for summarizing data from a specific location. Compressed files included in a ZIP, TAR or JAR file are first extracted out by an Expander component and then processed by the Summarizer. The Gatherer may also carries an embedded HTTP server so that system administrators can use a web-browser to control its operations and monitor its status.

One example of a method that may be performed by a Gatherer is the following:

Method Cyberspace Crawling

1. Starting from a collection of source URLs, e.g., `www.ibm.com`, perform a BFS-like search out of the hyperlinks;
2. When encountering a new page, build a summary for the page (Parts of information kept in a summary page that are important to the crawling method include a URL of a parent on the BFS-tree and all hyperlinks out of the page); and
3. Maintain a queue of the set of pages whose parents are found. (The gatherer may also maintain a hashtable of the set of URLs that have been summarized. If a URL referred by a hyperlink out of the current page has been processed, then it will not be put back into the queue).

In short, a gatherer performs the breadth-first-search (BFS) on a large web-graph (whose structure is not explicitly given in advance).

3 Collaborative Web Crawling: a High-Level Design

A collaborative web crawling (CWC) system uses more than one gatherer/crawler. These gatherers is organized to collaboratively explore a web-space, to generate summaries, and to store these summaries for future reference. In order to achieve the maximum efficiency in CWC, these are coordinated so that the load is balanced and the overhead is minimized. Coordination is achieved by partitioning a web-space (the URL space) into sub-space and assigning each subspace to a processor. Each processor is responsible to build summaries for those URLs contained within its assigned sub-space. During the construction of a summary, new URLs may be discovered by the Crawler. In this event, a process will keep processing those URLs belonging to its sub-space, including the new URLs if appropriate, route other URLs to proper processors.

To implement CWC, we use a method that allows the processors to communicate with each other. There are two types of information needed for the communication in CWC: *foreign URLs*, where a processor needs to send a URL given by a crossing hyperlink to the processor assigned to it; and *coordination signals*, used to re-map and load re-balance the system.

In our current design, we use TSpaces to support both foreign URLs and coordination signals. The TSpaces space, originally, proposed in a parallel programming language called LINDA, embodies three main principles:

- Anonymous communication;
- Universal associative addressing; and
- Persistent data

In a TSpaces, unstructured tuples may be posted to a universally visible TSpaces (by an “Out” command); tuples may be used and removed (using an “In” command); and they may be read (using an “Read” command) by any processor in the system. TSpaces have already been used in the IBM GCS as a general mechanism for connecting various distributed components.

In the future, we might use processor-to-processor communication mechanism for foreign URLs, and only use TSpaces for coordination signals. Figure 1 shows a high-level description of our current CWC architecture.

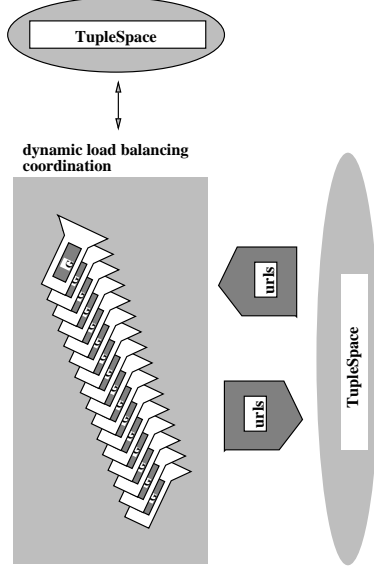


Figure 1: The architecture of CWC.

3.1 Partitioning and Load Balancing

A URL-space can be viewed as a large directed graph $W = (U, L)$, called a *web-graph*. In a web-graph, each web-page or URL defines a vertex in U . A URL is directly connected with another URL if the first one is linked to the page defined by the second one, defining a directed edge in L .

In CWC, this web-graph (or URL-space) is divided into sub-graphs (sub-URL-spaces) that are mapped among processors of CWC.

A k -way partition of a web-graph $W = (U, L)$ is a division of U into k subsets represented by U_1, U_2, \dots, U_k . Let $L_{i,j}$ denote the set of crossing links from U_i to U_j .

If a CWC system has k processors and U_i is mapped onto processor i , then the process i needs to examine $|U_i|$ URLs and send $|L_{i,j}|$ links to processor j during crawling, where $|U_i|$ and $|L_{i,j}|$ are the cardinality of sets $|U_i|$ and $|L_{i,j}|$. The communication of links from one processors to another is one kind of overhead in CWC.

To maximize efficiency, there are several load balancing measures that a CWC system may optimize.

1. **Work balance:** with respect to a partition U_1, U_2, \dots, U_k , the amount of work that processor i needs to perform is measured by the total time that processor i spends on retrieving pages from U_i , on generating the summary of these pages, and on processing new hyperlinks from these pages.

In CWC, different processors may have different machine speed, and depending on the site of the machine and the site of the server of the URL-pages, the accessing time of the pages will be different from processors to processors, and from URLs to URLs.

2. **Space balance:** with respect to a partition, the amount of disk space (memory space) needed by processor i is roughly proportional to the total size of the summaries that processor i needs to build.
3. **Minimize crossing links:** when a processor finds a page that is not in its partition, it may pass the information to the processor that is responsible for the page. These communication costs are among of the main overheads of collaborative crawling.

The main objective of our scheme is to optimize one or more of the above - matching processors and URLs to load balance and reduce processing overhead.

3.2 Challenges in collaborative team crawling

Graph partitioning has been a well-studied problem in parallel scientific computing and circuit design [8, 1]. However, partitioning algorithms developed previously do not work for optimizing CWC for numerous reasons.

First, the web-graph is not known (explicitly) to the partitioner of CWC in advance. In addition, the web-graph is very large and fast growing dynamically. In the process of crawling, more and more of this unknown graph may be explored, allowing additional information to be added to the explicitly given portion of the web-graph. Therefore, dynamic load balancing and load rebalancing are necessary.

Second, the web-graph is highly irregular. The web-graph may include multiple local expander subgraphs and internal vertices that have very large larger “in” and “out” degree. There may also be a lot of local sub-hierarchical structures that are highly connected with URLs leading outside the structures. There, our CWC system must includes methods for handling some special classes of subgraphs before we actually partitioning the web-graph.

Third, the web-graph is a directed graph, making it difficult to know the in-degree of each page directly. Moreover, web-pages are usually restricted to read-only status. This requires the CWC to coordinate among processors for checking whether a page has been visited, or a visit is necessary.

Fourth, and primarily affecting the system design of CWC, the URL-space hyperlink connections (locality) are very different from the network connections (locality). If gatherers/crawlers are located at different network sites, then partitioning the web-graph for achieving both hyperlink locality and network locality is desired.

4 Partitioning Web-Space

We present a dynamic web-space partitioning algorithm in a CWC system. It automatically mapping the web-space among processors in a CWC system and it is able to adjust to the dynamic change of the web-graph and accessing efficiency to achieve the best performance during a given run-time. It forms sub-domains that are spatially close, allowing each processor to map to a sub-domain to which it has the most efficient access. It also automatically form sub-domains that have more internal hyperlinks so that the communication overhead in parallel crawling is reduced. In general, our algorithm explores trade-offs between the hyperlink locality and network locality in dynamic load balancing and optimizes the map between web-subdomains and processors with different processing and communication capacity.

In addition, to minimize computation overhead for determining the processor that is responsible for a given URL, our partitioning algorithm does not use any semantic content of an URL-page, i.e., our method only uses information given in the URL name itself. Because crawling is an I/O intensive computation, it is expensive to decide which processor should examine a URL page by first fetching the page.

When the problem size is larger than the limit of memory of the processors, our method explores the trade-off between introducing redundant computation and accurate out-of-core computation. In addition, we have implemented methods to determine the fraction of the web-graph that has already been explored, by answering questions such as “are we halfway there?”.

4.1 Coarsening for a manageable structure over the web-graph

To cope with the large scale and dynamically changing web-graph with an implicit structure, and to reduce computation overhead, our algorithm uses *graph coarsening* [9] for partitioning the web-graph. Coarsening based partitioning methods have been successfully applied in the traditional setting for parallel scientific simulation [9].

The basic idea of graph coarsening is that given a graph $G = (V, E)$, a coarsened image $G' = (V', E')$ of G is created by vertex contraction. In other words, each vertex in V' corresponds to a subset of vertices in V , where a vertex of G' is a *super-node* of G . Similarly, each edge in E' , called a *super-edge*, corresponds to a collection of edges in G . Once the coarsened G' is created, G' can be partitioned and the partition can be mapped back to G . In general, good coarsened graph retains the locality of the original graph so that a good partition for the coarsened graph approximates a good partition for G . One advantage of the coarsening based partitioning is a reduction of the complexity of the web-graph because G' tends to be much smaller than G .

In CWC, graph coarsening may help to optimize both the network locality and the hyperlink locality,

- **Network locality:** If a processor is mapped to a sub-space that is “physically” local to the processor, then it can work most efficiently. This network locality of processors and web-page servers is characteristically defined by the network graph. To achieve this, URLs are clustered based on how close they are to each other in the network space. For example, all URLs from a server may be placed into the a same subdomain of the partition. This clustering by network space location is necessary because a domain name itself does not always directly give us any information of physical closeness. Nor is the Internet provider (IP) address of the URL the domain name always helpful in defining physical location. For example, in a domain called `ibm.com`, a `tokyo.ibm.com` subdomain may be physically far away from a subdomain named `almaden.ibm.com`. In comparison, a sub-domain of `parc.xerox.com` may be much closer to `almaden.ibm.com` in the network. On the other hand, `gcs.almaden.ibm.com` and `helpdesk.almaden.ibm.com` are very close in the network. Thus, proper expansion of the URL’s domain name hierarchy will gradually provides blocks of servers whose site are close physically.
- **Hyperlink locality:** Another way to optimize collaborative web crawling, other than balancing search load and optimize physical locality, is to op-

imize hyperlink locality, as defined by the web-graph, so that the partitioning reduces the amount of crossing links among processors. Because of limit space in TSpaces, or any other communication mechanism used, optimization of link locality reduces the number of tuples transferred to the TSpaces. In addition, the more tuples on TSpaces, the less efficient the TSpaces will be, creating communication bottleneck. For example, from our sequential experiments, the queue in BFS search can be very large.

Our web-graph partitioning scheme has three basic steps:

1. **[Coarsening]:** Apply coarsening to the web-graph introduce a manageable structure that has the following properties;
 - (a) the super-node retains the physical and hyperlink locality and the coarsened graph is much smaller than the original web-graph;
 - (b) given a URL, the super-node containing the URL can be determined efficiently;
 - (c) the number of hyperlinks and the average network distance between two super-nodes can be estimated efficiently; and
 - (d) the “relative” total amount of work needed to process web-pages in each super-node can be estimated efficiently.
2. **[Partitioning]:** Apply an quality static graph-partitioning algorithm to divide the coarsened web-graph and project the partition back to the original web-graph.
3. **[Dynamic Re-Coarsening and Repartitioning]:** When the load becomes unbalanced or when new processors are available or some existing processors become unavailable, re-coarsen the web-graph and repartition the coarsened web-graph based on the current statistical information about the web-graph generated by the crawling,

Our coarsening-based method dynamically forms good super-nodes, also referred as *superpages*, and contracted graphs defined by these superpages. We use the URL name hierarchy to define superpages to optimize localities and identification between URLs and superpages. The creation of superpages uses the information collected during the gathering phase to dynamically recognize which superpage should be formed for the next phase load rebalancing. Our partitioning algorithm then divides the coarsened web-graph defined by these dynamically created superpages and generates a mapping between processors and superpages to further optimize localities.

4.2 The URL name hierarchy

A URL usually contains information: a host domain name (e.g., `almadan.ibm.com`), protocol (e.g., `ftp`), port (e.g., `1080`), user ID, `uid`, (e.g., `steng`), password (e.g., `*****`), and URI, that is, the directory under a server, (e.g., `/docs/Slides/Tspaces/index.htm`). Currently, the host domain name and URI are used to define superpages as defined below. There is a natural hierarchical structure over URL names, referred as the *URL-hierarchy* or the *URL-tree*.

Figure 2 shows a part of the hierarchy defined by URL names.

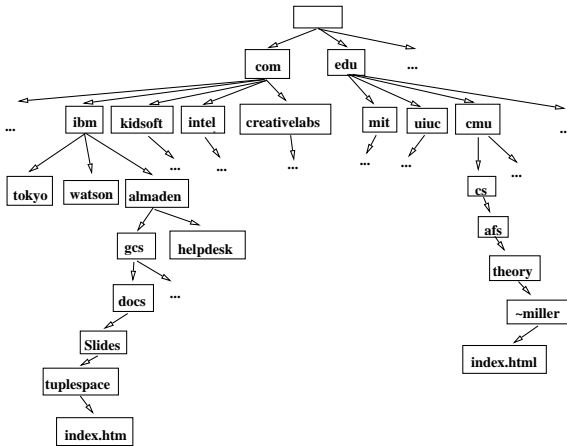


Figure 2: An example of URL-tree.

The first part of a URL name as shown in Figure 2 is its Internet host domain name. It is a hierarchical naming scheme where a name consists of a sequence of sub-names separated by a delimiter character, the period. Each section in the domain name is referred to herein as a *token*. For example, `com`, `ibm` and `tokyo` would be tokens of the domain name `tokyo.ibm.com`. The “top level” token of the domain name space consists of: `com` (commercial organizations), `edu` (education institutions), `gov` (government institutions), `mil` (military groups), `net` (major network support centers), `org` (organizations other than those above), `arpa` (temporary ARPANET domain), and `country code` (countries other than USA).

A domain name could be the name of a machine, of a sub-domain, or even of a user. The full domain name of a URL determines its Internet physical address. In general, there is a block structure (locality structure) relating domain names and IP addresses. For example, almost all servers in the sub-domain `cmu.edu` are closely located in Pittsburgh.

Further, the host domain names are associated with servers and are arranged in a tree structure that corresponds to the naming hierarchy. This hierarchy demarks sub-domains in our CWC, and defines the top section of

our URL-hierarchy. The URI also defines a hierarchy. So the URL-tree is formed by the domain name hierarchy followed by the URI hierarchy, where each URL is associated with a leaf in the URL-tree.

4.3 Superpage: an intermediate unit for dynamic load balancing

Suppose the CWC system has k Gatherer processors. Then we will divide the web-graph into k subgraphs, say W_1, \dots, W_k . Each sub-graph is mapped to a processor (e.g., W_i to processor i). If the processing load becomes unbalanced, repartitioning of the web-graph is required to achieve better performance. In this case, a new partition is W'_1, \dots, W'_k is formed and W'_i is mapped to processor i . For each i and j , processor i sends the URLs in $W_i \cap W'_j$ - currently located in its queue - to processor j . Similarly, processor j sends the URLs in $W_j \cap W'_i$ to processor i .

However, because the entire web-graph is not known during time partition nor repartition occur, other technique such as graph coarsening may be required because it is almost impossible to design a partition at the URL level with knowing the web-graph formation at the partition time.

Graph coarsening is used here to introduce an intermediate structure called *superpage* of URLs to generate a coarsened image of the web-graph. A superpage is a set of URLs which are “close” to each other according to the URL hierarchy. The set of superpages is formed dynamically in the gathering/crawling phase. A partition is a division of superpages into k sets.

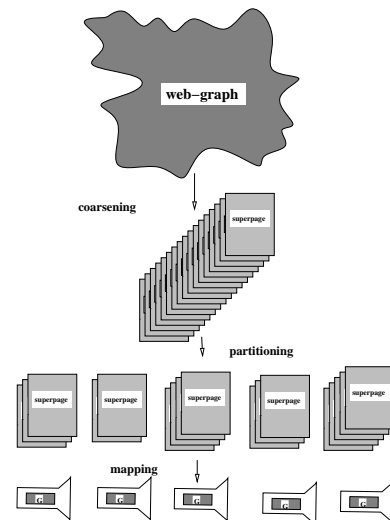


Figure 3: Superpages: coarsening based web-graph partitioning

Formally, we divide the web-graphs into a collection of superpages P_1, \dots, P_m , where m is usually larger than

k (the number of processors). From the original web-graph $W = (U, L)$, a new graph $C = \{P, S\}$ is constructed where $P = \{P_1, \dots, P_m\}$, a set of superpages, and $S = \{S_{i,j} : 1 \leq i, j \leq m\}$ where $S_{i,j}$ is the set of the hyperlinks from P_i to P_j . $S_{i,j}$ will be referred as the *super-hyperlink* from superpage P_i to P_j . The *weight* $w(P_i)$ of each superpage P_i is equal to the number of URLs in P_i or the estimated processing load of P_i . In other words, C is a coarsened graph of the web-graph W . Our partitioning algorithm is a two-level partitioning algorithm [9].

The basic idea for dynamic load balancing is to create superpages from the URL hierarchy, estimate the weight of a superpage by the number of URLs crawled or in the queue, and estimate the number of super-hyperlink in a similar way. The web-graph is then partitioned by partitioning the coarsened graph C over superpages. This allows quick decisions concerning which URL belongs to which partition once a partition is formed by first determining the superpage containing the URL by the URL-tree and then returning the partition number of the superpage.

4.4 Superpages Formation

The domain name hierarchy provides a good initial clustering of URLs into superpages concerning both physical and hyperlink localities. In general, there is a large volume of crossing hyperlinks among URLs of a same subdomain.

Given a URL, a sequence of tokens can be defined from its domain name and its URI. Tokens from the domain name is ordered from right to left, while the tokens for the URI is ordered from left to right. For example, `gcs.almaden.ibm.com/docs/Slides/Tspaces/index.htm` defines the sequence: (com, ibm, almaden, gcs, docs, Slides, Tspaces, index.htm). Following this sequence of tokens, we can reach to the leaf of this URL in the URL tree.

The superpages are defined by a “frontier” of the URL tree when the tree is “expanded” from the root. For example, imagine initially we have only one superpage, the root of the URL tree; the frontier of the URL-tree is just the root. Any node of the URL-tree may be expanded either by singletons or by intervals: For example, suppose the root is com. The singleton containing `ibm` defines an expansion of the root, that is, the child of the root whose token is `ibm`. Similarly, a singleton expansion of `ibm` with `almaden` as the token can be made.

In an interval expansion, we choose a sequence of ordered strings, and divide the children of the node of the tree into intervals. For example, for the root `com`, ordered strings (N, Kidsoft, H, creativelabs) can be used to divide the children of the root into intervals whose next token is in [N or larger], [Kidsoft,

N), [H, Kidsoft), [creativelabs, H), [smaller than creativelabs], respectively.

The singleton expansion extends the URL trees naturally; while the intervals expansion clusters the children of a node in the frontier of the tree. In our scheme, an interval is not extended beyond a level, where the intervals will always be a leaf of the tree. Once an expansion has occurred, the node at the frontier is used as the superpage. Figure 4 shows an example of an expansion and its frontier.

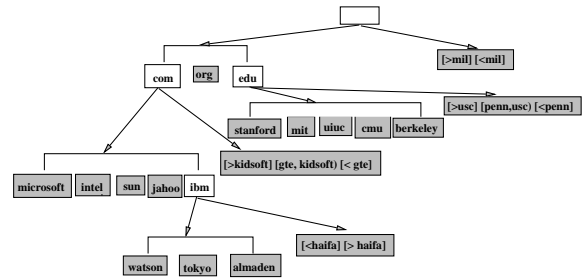


Figure 4: Expansion and frontier.

In CWC, a user may provide an initial expansion. Our algorithm automatically generates a partition from this expansion. Once the gathering starts, each processor will maintain some statistical information in the partial URL tree that is currently used for the defining superpages. The statistical information includes a set of tokens in each node which are not currently used in the singleton expansion, but which may be the candidates for the next round singleton expansion. We also maintain a random sample of all visits of a node; it will be used for re-defining an interval expansion of the node. In addition, we record the number of visits to each internal nodes of the current partial URL tree; it will be used to estimate the weight of each new and old superpage. As discussed above, the estimated weights will be used for load rebalancing.

Given these information and a selected threshold for the superpage information size, we can automatically generate the set of new superpages by proper singleton and interval expansions, as discussed in Section 5. Figure 5 shows an example of new superpages created from the frontier given in Figure 4.

4.5 Superpage Partitioning

Using superpages permits an addressable structure to be established over the web-graph, where every URL has a superpage “number” or address. Even though the precise number of URLs in a superpage may not be known, or how a superpage is connected with other superpages, information gathered can be used to estimate these measures during the processing of crawling/gathering. Further, superpages can be used to reduce

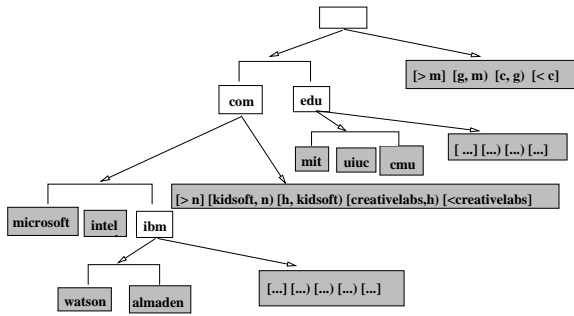


Figure 5: Dynamic superpage creation.

the partition problem of a large scale unknown graph to a partition problem of a reasonable sized known graph as shown below. We present algorithms for mapping superpages onto processors to balance/re-balance the processing load, and to optimize the match between processors and superpages.

During crawling/gathering, each processor obtains and maintains some statistical information on the average access time and processing time of a URL page from a superpage. In the simplest form, we combine these two pieces of time into a single parameter hereafter referred to as a *processing rate*. Hence, for m superpages and k processors, the processing rate information can be viewed as a k by m table, $R = [r_{ij}]$ where r_{ij} is rate that processor i can access/process URL pages in superpage j .

In addition, all processors collaboratively bookkeep the number of URLs visited in each superpage and to be visited in each superpage and also maintain the size of super-hyperlink among these superpages. In the partition phase, we assemble an array $w = [w_i]$, where w_i is the number of URLs visited at a given time that belong to superpage i , and $S = [S_{i,j}]$ where $S_{i,j}$ is the super-hyperlink from superpage i and superpage j . We will consider several versions of the partition problem.

- Given a collection of m superpages $\{P_1, \dots, P_m\}$ with weights $\{w_1, w_2, \dots, w_m\}$, and k processors whose access rates to superpages are given in table R , divide the superpage into k subsets U_1, \dots, U_k that minimizes $\max_i^k T_i$ where

$$T_i = \sum_{P_j \in U_i} w_j / r_{i,j}.$$

In other words, this version will find a partition that not only balances the work-load of each processor, but also optimizes the match between the superpages and processors by trying to assign each superpage to a processor which has the most efficient processing rate. This version is applicable to the case when it is a secondary concern to minimize the amount of super-hyperlinks that across

partitions. In fact, previous sequential experiments showed that this is often the case in web-crawling.

- In addition to a collection of m superpages $\{P_1, \dots, P_m\}$ with weights $\{w_1, w_2, \dots, w_m\}$, and k processors whose access rates to superpages are given in table R , we are given one more table c of rates $c = [c_i]$ where c_i is the rate for communicating (sending and receiving of) URLs to other processors. In this case, the partitioning problem is to divide the superpage into k subsets U_1, \dots, U_k that minimizes $\max_i^k T_i$ where

$$T_i = \sum_{P_j \in U_i} (w_j / r_{i,j} + B_i / c_i),$$

where B_i is the number of hyperlinks that are to or from superpages in U_i that are not in U_i itself.

In other words, this version finds a partition that not only balances the work load of each processor and optimizes the match between the superpages and processors, but also minimizes the communication overhead,

5 Dynamic load balancing

In this section, we present methods for dynamically creating superpages as well as partitioning superpages. Given an expansion of the URL-tree, we define the set of superpages from the frontier of the expansion. These superpages are then partitioned and distributed across the Gatherer processors. Each Gatherer processor initially “sees” the same expansion and hence set of superpages.

Each Gatherer then runs the sequential program of GCS. When it encounters a new URL, it first checks with the current expansion of the URL-tree to determine the superpage that contains the URL. It locates the partition that contains the superpage. If the superpage is local to the Gatherer itself, it checks with the already-visited pool to decide whether this URL has been processed. If not, the new URL will be added to its local queue. If the URL belongs to a superpage which is mapped to some other processors, the Gatherer forwards it to the proper processor. In our current implementation, this is done by posting URL address and the ID of the assigned processor to the Tspaces. Every Gatherer has an observer thread to receive the URLs sent from other Gatherers and to insert them in its local URL queue, if appropriate.

When a URL is pushed down the current expansion of the URL-tree, a set of statistical data stored at each node of the tree is updated. This data will be used further for dynamic load balancing.

5.1 Hot Tokens and Samples Tokens

Whenever a URL passes through a node in the URL-tree, the number-of-visits counter for the node is increased by one. Each Gatherer only updates its own copy of the expansion. But, if those numbers are added together among all copies of the expansion, the number of URLs that belong to the subtree of a node in the expansion can be obtained.

To assist dynamic load balancing, we maintain two other pieces of information at each node in the expansion: hot tokens and sample tokens. When a URL reaches a node in the expansion tree, its next token in the token sequence of the URL name is examined. If the next token matches to a token of an singleton children expansion, it is pushed down to the next level of the expansion tree and the number-of-visits counter is increased. The unmatched tokens are divided into two classes: hot tokens and cool tokens. We maintain a hot token hashtable to bookkeep the top 50-200 of the most often used unmatched tokens and their counters. An unmatched token not in this hashtable is called a *cool token*, and we maintain a random sample of 50-200 unmatched tokens by using an on-line sampling algorithm.

5.2 Dynamic superpage creation

Dynamic load balancing may be triggered in our current implementation in one or more of the following circumstances: a processor is too idle; a new processor joins the crawling team; a processor decides to quit the crawling team; or the physical locality is poorly achieved in the partition. Other events specified by the user may singularly or collectively trigger load rebalancing also.

Once load re-balancing is to be performed, the Gatherers use the Tspaces to reach a consensus and select a “leader” processor to perform the repartition. Every other processor will send its copy of the expansion tree to the leader. The leader will merge the information collected at each node in the expansion tree. The resulting copy of the expansion tree will maintain all the statistical data about each superpage as well as about each internal node in the expansion tree such as the number-of-visits counter, the set of hot tokens, and the set of sample tokens.

The leader uses the following superpage creation algorithm at each node of the expansion: Let k be the number of processors and δ be a parameter that determines a threshold of the weight of superpages, typically, 20 or more. Let W be the number-of-visits to the root of the expansion tree. W is equal to the number of URLs that have been processed or put in the queue, which is an estimation of the total weight of the root. Let $\Delta = W/(k\delta)$.

Algorithm Superpage Creation(Δ)

1. If there is a hot token whose count of visits is at least Δ , perform a singleton expansion with that token. The weight of the singleton expansion is equal to its number-of-visits count.
2. Re-sampling the remaining hot tokens into the cool token samples.
3. Let the I be ratio of the total number of cool-token visits to that of Δ . I will be the number of interval expansions created; and
4. Choose I evenly spaced tokens in the cool-token-samples and create I interval expansions.

The above **Superpage Creation** method is applied to every node in the expansion tree, and generates a new frontier to the expansion tree that is used to define the set of new superpages for the next round of gathering/crawling.

5.3 Superpage partitioning algorithms

In order to optimize the physical locality among URLs, the average access rate of each processor to the servers in each superpage is maintained during the crawling process. These data will be sent to the leader to assemble the access/process rate table R mentioned in Section 4.5.

We now consider the first version of the partition problem given in Section 4.5. This version is NP-hard even when $r_{i,j} = 1$ for all i and j . Mathematically, there is a linear-programming based approximation algorithm that solve this problem to within a factor of 2 of the optimal. In our current CWC system, we use the following greedy algorithm.

Algorithm Superpage Partition I

1. Sort the superpage in an non-increasing order in their weights, where $w_1 \geq w_2 \geq \dots \geq w_m$.
2. Let $T_i = 0$ and $U_i = \emptyset$ for all $1 \leq i \leq k$.
3. for ($j = 1; j \leq m; j++$)
 - (a) Let $\Delta_i = w_i/r_{i,j}$ for i in the range $1 \leq i \leq k$.
 - (b) Add P_j to U_s with the s that has the smallest $T_s + \Delta_s$; Let $T_s = T_s + \Delta_s$.
4. Return U_1, U_2, \dots, U_k .

The above partition can be improved by applying some local-improvement methods. For example, one way is to reduce $\max_i T_i$ by moving a superpage from the U_s with the maximum T_s to some other set.

In the first version of the partitioning problem, the amount of crossing hyperlinks among superpages in different partitions was not included as a quality measurement. However, it is a reasonable simplification for practical implementation for the following reason: Graph theoretically, the quality of the best partition is getting worse if the connection of the graph is getting higher. Thus, in the case when a graph (such as the coarsened web-graph) is highly connected, it is more important to optimize load balance than minimizes the volume of crossing connections. In CWC, we apply graph partitioning on a coarsened image of the web-graph defined over superpages. This coarsened graph much more highly connected than the original web-graph. The original web-graph or most of its subgraphs (e.g., subgraphs defined by web-pages only within a particular domain such as `cmu.edu` or `ibm.com`) are already quite highly connected, in the sense that they are expander-like. Therefore, the coarsened graph defined on superpages is very highly connected. Our scheme explores physical and hyperlink localities in superpage creation and focus more on load balancing in partitioning superpages.

In the case when it is important to consider the impact of crossing hyperlinks among superpages, we use the the second version of the partition problem. This optimization problem is again NP-hard even when $r_{i,j} = 1$ for all i and j and $c_i = 1$, and there is a linear-programming based approximation algorithm to solve this problem. In practice, we extend our previous algorithm.

Algorithm Superpage Partition II

1. Sort the superpage in an non-increasing order in their weights, where $w_1 \geq w_2 \geq \dots \geq w_m$.
2. Let $T_i = 0$ and $U_i = \emptyset$ for all $1 \leq i \leq k$.
3. for ($j = 1; j \leq m; j++$)
 - (a) Add P_j to U_i may increase the time for all processors, including T_i , where an increase on T_h is $\Delta_{i,h}$.
 - (b) Add P_j to U_s with the s that has the smallest $\max_h T_h + \Delta_{s,h}$, where $T_h = T_h + \Delta_{s,h}$; and
4. Return U_1, U_2, \dots, U_k .

The partition can also be further improved by various local-improvement methods.

6 Hot Pages and Boundaries of a Partition

A web-graph is generally highly irregular and highly connected. Many web-pages have very large in- and out-degrees, and there may be many densely connected

and bi-connected subgraphs (“communities”) of pages. The diameter of the web-graph, the number of hyperlinks connecting any page to any other page, is a much slower growing function in the total number of web-pages. These properties makes partitioning and load balancing the web-graph challenging.

In this section, we present a method that dynamically recognizes these special pages, substructures, and highly connected communities, and improves the quality of partitioning and reduces the amount of communication required for collaborative web crawling.

In crawling, two structures are maintained: the hit-hashtable of those URLs that have already been visited, and a queue of those URLs who parents have been visited. When summarizing a web-page, each out-going URL link is examined. If the URL has already been visited, then it URL is dropped, otherwise, it is insert into the queue. Preferably, before it is inserted into the queue, the URL is checked as to whether it has already been put into the queue at previous stages. The queue grows much faster than the hit-hashtable (more than 15 times by our sequential experiments). In fact, it may grow too fast to b reasonably stored in the memory. If this is true, then the queue is maintained in storage, and only a window of the queue remains in memory.

This introduces an challenging engineering problem. Because the queue grow so quickly, the queue may become too large for quick indexing and checking, thereby requiring expensive out-of-core computation. The the new URL will be inserted at the end of the queue. The result is that there may be more than one copy of a URL in the queue. For example, if a page has M in-links, then, its could show up in the queue M times.

In distributed crawling, the problem of checking which URL has been processed and queued becomes enormous. As a simple approach, each processor maintain its own hit-hashtable and queue for its respective partition. If a URL belong some other processors’ partition, it is posted on the Tspaces. The processor that extracts the posted URL from Tspaces checks with its own hit-hash and perhaps its queue. If a URL is linked to by M URLs from sub-domains other than the sub-domain that contains the URL, then it will be sent to Tspaces M times, which will worsen the fact that Tspaces is the communication bottleneck.

As an alternative method, each processor maintain a communication queue, c-queue, for those URLs to be sent to Tspaces, and a communication hit-hashtable, c-hit-hashtable, for those URLs that have been posted. Before posting a URL to Tspaces, the processor will check whether it has already in the c-hit-hash or c-queue. This reduce the load of the Tspaces, at the expense of the local space and time of each processor. For the same reason, the c-queue might be too large to stay in the main memory.

We solve this problem with a concept called *hot-pages*. Each Gatherer processor maintains an in-core hashtable that only stores hot-pages, that is, those URLs that have a potential large in-degree. When the processor encounters a new URL, it checks against the hot-page-hashtable. We have developed and implemented a dynamic mechanism to determine hot-pages and to keep the hot-page-hashtable small and efficient, assuring that it can be stored in the memory.

The removal of certain set of highly connected sub-graphs will dramatically reduce the connectivity of the original web-graph. These sub-graphs form the boundaries of the partition. The idea of hot-page is the first step in this investigation. In this setting, the special sub-graphs are pages that have large in-degree. Our experiment reports very good improvement of using this idea in the reduction of the queue size and in reduction of the amount of crossing links that need to be sent to Tspaces.

7 Team crawling with a pool of light-weight helpers

In many organizations, there are a large collection of computers that will be idle during a certain period of time, e.g., at night. These machines may be properly used to improve CWC. In this section, we present a design on how to make a sensible use of these machines.

The owner/user of a machine should have full rights to decide at what time and for how long he/she would like contribute the machine for crawling. Further, the owner usually does not want any additional inconvenience. In particular, no additional side-effect should result involving their machines. So in our scheme, these machines does not store crawling information on their local disks. In addition, because the owner can decide when to withdraw a machine from participation, the crawling program can not have a local queue that needs to be processed when the machine terminates its participation. As such, these machines are referred as *light-weight* machines for crawling. When these machines are used for crawling, dynamic load balancing is not performed each time a light-weight machine joins or withdraws from the crawling team. We develop a two-level architecture for collaborative web crawling with light-weight machines. Figure 6 helps to illustrate this architecture.

There are two set of crawlers: regular crawlers and light-weight crawlers. The regular crawlers are coordinated by their own TSpaces domains for crossing hyperlinks and for partition coordination. The system as shown in Figure 6 has two additional TSpaces domains, URLTS and SummaryTS, for communicating with light-weight machines. The Tspaces domain URLTS is for URLs and SummaryTS is for Summaries. Regular crawlers can send URLs from its queue to URLTS for

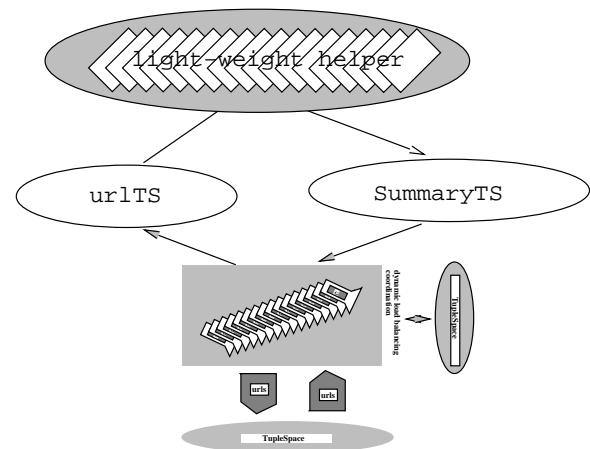


Figure 6: An architecture for using light-weight machines.

light-weight machines. The light weight machines that join crawling retrieve URLs from URLTS, gather pages, generate summaries for these pages, and send the summaries to SummaryTS. The regular crawlers then retrieve their own summaries from SummaryTS, extract a set of hyperlinks from the summaries, and process them as they normally do. In this design, the light-weighted machines are used only as occasional helpers.

At the implementation level, we can use Applet. On the machine that supports URLTS and SummaryTS, we set up the Applet for the light-weight helpers to participate. The Applet program will read from URLs from URLTS, use the machine cycle of these light-weight machines to convert URLs to summaries. However, the current Netscape security model does not allow the Applet program to have all network access. Fortunately, the new security model of JDK 1.1 (and already of Hot-Java) allows a browser to grant the network access to light-weight machines. Because in our architecture, the light-weight crawlers and regular crawler are inside the same firewall, it is feasible to make this configuration.

The amount of changes on the CWC system is quite small. The partitioning and load balancing methods are unchanged with the addition of these helpers. The load still need to be balanced, for otherwise, a regular crawler will have too many of its summaries in SummaryTS.

In an distributed environment, for example, if we have 10 regular members at IBM Almaden and 9 at IBM Watson, separated URLTSs and SummaryTSs can be used for the light-weight crawlers so that the Watson's light-weight crawlers only works for regular crawlers in Watson, while the light-weight crawlers at Almaden only help regular crawlers at Almaden. The average amount of light-weight help can be taken into

consideration in dynamical load balancing among the regular crawlers.

8 Future Research

The use of hot-pages, from a graph algorithm view point, makes a graph easier for partitioning. As a long term research project, we will investigate the following techniques for partitioning the web-graph:

1. recognize a collection of sub-graphs that are themselves highly connected, and are highly connected to and from other parts of web-graph;
2. make this collection of special subgraphs a separate class and either share them among all processors, or partition them among all processors;
3. apply our technique to partition the remaining web-graph; and
4. design algorithms to handle URL links to and from the special class of subgraphs to reduce the crawling overhead.

An alternative approach of parallel crawling is to use a large indexing database which serves as a central shared space for all communications and summaries [5]. Our approach for distributed collaborative web crawling is more scalable in part because crawling an I/O intensive process, and our scheme does not rely on a central system, such as the index database. More real-life experiments will be performed to confirm this.

Currently at IBM Almaden, a new sequential version of GCS have been implemented. This new program uses a database (DB2) to support the queue for visited and to-be-visited URLs. Our scheme is planned to be incorporated into the new version of GCS.

Acknowledgments We would like to thank Norm Pass for his support and for his suggestion of using light-weight crawlers.

References

- [1] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: a survey. Technical Report of Department of Computer Science at UCLA, 1996.
- [2] C.M. Bowman, P.B. Danzig, D.R. Hardy, U. Manber and M.F. Schwartz, The Harvest information discovery and access system, *Computer Networks and ISDN Systems* 28 (1995) pp. 119-125.
- [3] Matthias Eichstaedt, Qi Lu, Shang-Hua Teng Parallel Profile Matching for Large Scale Webcasting Lecture Notes in Computer Science Springer-Verlag, pp 17-28, 1998
- [4] IBM: all searches start at Grand Central, Network World, November 11, 1997 front page
- [5] <http://www.hotbot.com/>
- [6] Information on the Fast Track, *IBM Research Magazine*, Vol. 35, No.3, 1997, pages 18-21.
- [7] <http://www.pointcast.com/>
- [8] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Automatic mesh partitioning. In A. George, J. Gilbert, and J. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, IMA Volumes in Mathematics and its Applications. Springer-Verlag, 1993.
- [9] S.-H. Teng. Coarsening, sampling, and smoothing: elements of the multilevel method. The IMA Volumes in Mathematics and Its Applications, R. Schreiber ed. Springer-Verlag, 1998.
- [10] P. Wyckoff, D. Ford, and T. Lehman. IBM TSpaces. *IBM Systems Journal*, Aug, 1998