PAGE-TO-PROCESSOR ASSIGNMENT TECHNIQUES FOR PARALLEL CRAWLERS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Ata Türk

September, 2004

ABSTRACT

PAGE-TO-PROCESSOR ASSIGNMENT TECHNIQUES FOR PARALLEL CRAWLERS

Ata Türk
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2004

In less than a decade, the World Wide Web has evolved from a research project to a cultural phenomena effective in almost every facet of our society. The increase in the popularity and usage of the Web enforced an increase in the efficiency of information retrieval techniques used over the net. Crawling is among such techniques and is used by search engines, web portals, and web caches. A crawler is a program which downloads and stores web pages, generally to feed a search engine or a web repository. In order to be of use for its target applications, a crawler must download huge amounts of data in a reasonable amount of time. Generally, the high download rates required for efficient crawling cannot be achieved by single-processor systems. Thus, existing large-scale applications use multiple parallel processors to solve the crawling problem. Apart from the classical parallelization issues such as load balancing and minimization of the communication overhead, parallel crawling poses problems such as overlap avoidance and early retrieval of high quality pages. This thesis addresses parallelization of the crawling task, and its major contribution is mainly on partitioning/page-to-processor assignment techniques applied in parallel crawlers. We propose two new page-to-processor assignment techniques based on graph and hypergraph partitioning, which respectively minimize the total communication volume and the number of messages, while balancing the storage load and page download requests of processors. We implemented the proposed models, and our theoretic approaches have been supported with empirical findings. We also implemented an efficient parallel crawler which uses the proposed models.

Keywords: Parallel crawling, graph partitioning, hypergraph partitioning, page assignment.

ÖZET

PARALEL AĞ TARAYICILARI İÇİN SAYFA ATAMA YÖNTEMLERİ

Ata Türk
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2004

On yıldan kısa bir süre içerisinde, Web (World Wide Web), bir araştırma projesinden, toplumumuzun her yüzünde etkili, kültürel bir fenomene dönüşmüştür. İnternetin popülaritesindeki ve kullanımındaki artış, İnternette bilgi aramayı sağlayan tekniklerin etkinliklerinde de bir artışa neden olmuştur. Ağ tarama bu tür tekniklerden birisidir. Bir ağ tarayıcı, genellikle arama motorlarını ve ağ depolarını beslemek için Web sayfalarını indiren ve kaydeden bir programdır. Bir ağ tarayıcısının faydalı olabilmesi için, kısa bir süre içerisinde yüksek miktarlarda bilgiyi tarayabilmesi gerekmektedir. Genellikle, etkin bir tarama için gerekli olan yüksek indirme hızlarına tek işlemcili sistemlerde erişilinemez. Bu yüzden, günümüzdeki büyük çaplı uygulamalar, ağ tarama problemini çözmek için çok işlemcili paralel sistemleri kullanırlar. Paralel ağ tarama, eşit yük dağıtımı ve haberleşme hacminin ya da mesaj sayısının azaltılması gibi bilinen problemlerin yanında, çakışmaların önlenmesi ve yüksek kalitedeki sayfaların erken taranması gibi problemlerin de çözümünü gerektirir. Bu tez, ağ tarama işleminin paralelleştirilmesi konuludur ve temel olarak ana katkısı paralel ağ tarayıcılarında sayfaların işlemcilere atanması işlemindedir. Bu tezde, çizge ve hiper-çizge modellerini bölümlemeye dayanan, iki yeni sayfa atama yöntemi önermekteyiz. Yöntemlerimiz, toplam haberleşme hacmini ve toplam mesaj sayısını azaltırken, işlemci başına düşen depolama yükünü ve taranması gereken sayfa miktarını dengelemektedir. Tez sırasında önerdiğimiz modeller uygulamaya dönüştürülmüş ve teorik yaklaşımlarımızın doğruluğu deneysel sonuçlarla kanıtlanmıştır. Ayrıca önerilen yöntemleri kullanan etkin bir ağ tarama programı yazılmıştır.

Anahtar sözcükler: Paralel ağ tarama, çizge bölümleme, hiper-çizge bölümleme, sayfa atama.

To my mother,

Acknowledgement

I thank to my advisor Cevdet Aykanat for his guidance and motivation. Algorithmic and theoretic contributions of this thesis have rooted due to our long discussions and his insightful comments.

I would like to say my special thanks to Berkant Barla Cambazoğlu, who not only proposed the original idea that initiated the works in this thesis, but also contributed continuously through the design and development of the studies we explain in this thesis.

I would like to thank to Cevdet Aykanat, Ozgür Ulusoy, Uğur Güdükbay, Berkant Barla Cambazoğlu, Bora Uçar, Eray Özkural and Tayfun Küçükyılmaz for reading and reviewing the draft copy of this thesis.

I am grateful to all of my friends and colleagues for their moral and intellectual support during my studies. Without the support of Kamer, Çağdaş, Bora, Barla, Hüseyin, Özer, Aylin, Oğuz, Engin, Sengör and my other friends whose names I have not mentioned, I could not have endured this thesis. Special thanks goes to my old friend Kamer Kaya, who has been by my side in every aspect of life.

I would like to express my gratitude for my family for their persistent support, understanding, and love, without which this thesis could not have flowered. Especially for my mother, I would like to have a way of expressing my feelings of gratitude and joy for being so lucky by being the son of such a wonderful woman. Unfortunately, I can not express such feelings with either written or spoken words.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The World Wide Web is by far among the most successful *software creatures* living in the information society. It continuously grows and evolves according to the changing technologies, and thus, avoids being obsolete. Due to its enormous growth, millions of people use specific tools like search engines and Web portals to search the Web. In order to provide up-to-date and thus accurate results, search engines try to keep a local and fresh replica of the publicly indexable Web pages. Achieving such a task is challenging by all means as Web pages tend to increase in number and their contents tend to change quickly. A survey on the size of the public Web [5] claims that, as of June 2002, the number of public Web sites is beyond three million, and they contain approximately 1.4 billion Web pages. Today, in 2004, the Google search engine claims to have indexed more than 4 billion pages [3]. A study by Cho and Garcia-Molina [1], analyzes the rate of change in the Web. In this study, which is based on crawling the same 720000 pages each day for a four month period, it is stated that 40% of the pages among 720000 changed within a week, 23% of the pages that fall into *.com* domain changed daily and it takes about 50 days for 50% of the pages to change or to be replaced by new pages. In order to achieve a download rate that can harvest *World-Wide* data with sufficient freshness, usage of multiple parallel processors turns out to be a necessity.

The parallel crawling problem has been addressed by people from different scientific communities varying from Information Retrieval to Mathematics whereas contribution from parallel computing community is relatively restricted. Even so, we believe an analysis from the perspective of the parallel computing communities may provide additional insight to the implications of parallel crawling. In parallel computing problems, assignment of tasks to processes plays a key role on the computation and communication costs of the overall parallel system. In parallel crawling problem, this assignment is mostly done by hash functions that take page URLs as seed and determine the responsible process for a page according to the hash values generated by these hash functions. We believe that instead of such a random assignment, a new assignment technique that will place pages that are highly connected –i.e. that have hyperlinks pointing to each other– together will provide reduced communication costs between the crawling processes and thus will perform better.

This thesis proposes two new task-assignment or page-to-processor assignment techniques, depending on graph partitioning and hypergraph partitioning. The efficiency of these techniques can be enhanced by applying them in different granularities, i.e., in page granularity or in site granularity. Having proposed these models, the usability and efficiency of the models are validated by conducting a series of experiments analyzing our models, comparing them with the traditional methods, and implementing a parallel crawler.

The structure of the thesis is as follows. In Chapter 2, we present the general crawling challenges and the architecture of our experimental crawler *ABC*. In Chapter 3, we explain parallel crawler architectures and advantages and difficulties of parallel crawling. In Chapter 4, we propose two models based on multi-constraint graph partitioning for page-to-processor assignment and site-to-peocessor assignment. The models only differ in the granularity they apply on, i.e., page-based or site-based. Chapter 5 proposes two models based on multi-constraint hypergraph partitioning. Again, the models differ in the granularity they apply. In Chapter 6, we provide an explanation of our testbed and present experimental results verifying the validity of the proposed models. Finally, we conclude and analyze future prospects in Chapter 7.

Chapter 2

Crawling Problem

In parallel to its growing size, structural complexity of the Web has reached to such a level that, without the help of specific information retrieval tools, a Web user who surfs among Web pages by just following hyperlinks among pages will most likely fail in locating his/her target information. In order to compensate this inefficiency in Web structure, state-of-the-art search engines and Web portals have started to serve as an interface between the users and the Web content, and thus, they have gained enormous importance and attention as they have turned into irreplaceable tools for Web users. Search engines and Web portals provide their services by analyzing the Web content and graph structure of the Web. In order to provide accurate and useful information to their users, search engines and Web portals have to form huge Web repositories, which ideally cover the whole Web. Furthermore, they have to refresh their repositories in concordance with the Web page changes. These repositories are formed through the usage of *Web crawlers*. Web crawlers, sometimes called as spiders, robots, bots, or wanderers, are tools which collect data for search engines. Many crawlers wander freely within the Web in order to retrieve data for major search engines and Web caches today. Within this chapter, our focus will be on providing an understanding of crawlers, the basic crawling algorithm, generic problems to be addressed in crawler design, and architectural components of an agent of our experimental crawler *ABC*.

2.1 Introduction

A crawler is a program which retrieves and stores Web pages by traveling over the Web graph using the hyperlinks among pages. The main algorithm for a crawler is deceptively simple. A basic crawler starts with an initial set of seed URLs to crawl, adds those URLs to its queue, downloads URLs within its queue, extracts the hyperlinks within downloaded pages, and then adds the URLs extracted from these hyperlinks (those of which have not already been added) to its queue. A Crawler simply continues to run until its URL queue gets empty. A variant of the stated algorithm is given in Algorithm 1.

In traditional crawler designs, the Algorithm 1 is run until a desired number of pages is crawled. In such systems, the crawling of a new collection is done from scratch whenever a fresh collection is required. A *crawling session* or *crawling cycle* is a term referring to the process of crawling a desired number of pages and then stopping the crawl.

Algorithm 1 Basic crawler algorithm.

```
Require: $Q$ is a queue of URLs
     $S$ is an initial set of seed URLs
     $H$ is a set of hyperlinks
 1: $Q \leftarrow S$
 2: $H \leftarrow \emptyset$
 3: while $| Q | \neq \emptyset$ do
 4:     $h \leftarrow$ DEQUEUE($Q$)
 5:     retrieve page $p$ pointed by hyperlink $h$
 6:     store page $p$
 7:     add every hyperlink $h \in p$ to set $H$
 8:     for all $h \in H$ and $h \notin Q$ do
 9:         ENQUEUE($Q, h$)
10:     end for
11: end while
```

The simplicity of Algorithm 1 is deceptive in the sense that implementing a working, basic crawler requires addressing many more challenges than simply implementing such an algorithm. In particular, a crawler must be polite, that is: it must not overload Web sites, it must be able to handle huge volumes of data, it must have a good coverage, it must meet the freshness requirements of the Web to provide accurate data, and it must pay attention to details such as: handling dynamic pages and avoiding *black holes* (spider traps). These challenges are explored in depth in Section 2.2.

Implementing an efficient crawler is a complex task. Hence, it is useful to divide the functionality of a crawler into basic components in order to reduce the overall design complexity and enhance modularity. A probable list of the components required to implement an efficient crawler is given below:

- A component for avoiding server overloading,
- A DNS resolving component for URL-to-IP-address resolution,
- A fetcher component which retrieves pages from the given IP addresses,
- An HTML parsing component for extracting the URLs from downloaded pages,
- A checker component for checking whether a given URL was discovered before,
- A component for storing downloaded page contents into page repositories,
- A component for checking *robots.txt* files to see whether discovered URLs are allowed by site admins.

This list can be extended easily. A crawler designer can follow various design choices while implementing these main components, such as merging these components into a single component or further dividing some complex components into smaller components. Section 2.3 provides a detailed analysis of the design choices that we have made for the components of our experimental crawler *ABC*, as well as providing information about its general architecture.

2.2 Challenges in Crawling

Design and implementation of a Web crawler compatible with the existing search engine crawlers is a challenging task. We have listed and analyzed some of the major difficulties that one has to overcome to accomplish this complex endeavor below:

- **Politeness:** It is vital for general purpose Web crawlers to achieve high download rates, but, while doing so, a crawler must avoid overloading the Web servers it visits. It is natural for a site admin to ban a Web crawler from accessing his/her site given that the specific crawler consumes an unacceptable portion of the site's resources. Thus, it is expected from a crawler not to download more than one Web page from the same host concurrently, and if possible limit the amount of resource consumption from a server while crawling. Furthermore, a crawler is expected to download only the Web pages allowed by site admins. Many Web crawlers offer this polite service through obeying The Robots Exclusion Protocol [16]. The Robots Exclusion Protocol allows site admins to indicate which parts of their sites must not be visited by a robot. This is achieved by placing a special format file named *robots.txt* in their sites' base directory. A "polite" crawler obeys the Robots Exclusion Protocol and also avoids overloading sites by not making more than one Web page request from the same site simultaneously.
- **Huge data structures:** There are certain types of intermediate and temporary data that each crawler must keep and use through a crawling session. Many times, if careful selection of data structures is not made, the size of the data structures keeping these intermediate data may exceed the size of the available memory. Some examples to these intermediate data might be:
  - A to-be-crawled URL list: Each crawler must keep a list of URLs that it is going to visit.
  - A crawled-URL list: Avoiding to crawl the same URL twice is vital for an efficient crawler design. Thus, each crawler must keep a list of URLs that have been crawled or identified previously. Whenever a

URL is extracted from a downloaded page, it must check this list to avoid duplicate crawls.

- An IP and domain name list: Keeping a list of resolved domain names and their respective IP addresses in memory avoids duplicate DNS queries to be made by the crawling system.
- A crawled-page content-hash list: There are many duplicate-content pages with different URLs within WWW. A crawler, after downloading such pages, should not store those pages in its repository, as this would cause a waste in storage resources. In order to avoid storing duplicate-content pages, keeping the hash of the stored pages and comparing the hash of newly crawled pages with the stored page hashes is necessary.

• **Freshness:** A document crawled by a crawler is considered to be *fresh* if its content had not changed during its last crawl. Let $\mathcal{W}$ be the set of all pages crawled by a crawler in its last crawling session. Let $\mathcal{C}_i$ be the set of pages in $\mathcal{W}$ whose content changed during the time period between their last crawl and time t_i. *Freshness* of a crawler at time t_i can be described with the formula: $\mathcal{F}(i) = \frac{\mathcal{W}-\mathcal{C}_i}{\mathcal{W}}$. Ideally one would like to keep the freshness value close to one. To achieve such a goal, the Google search engine crawls more than 4 billion pages once a month [3] as well as keeping a list of hand-selected pages which are determined to be changing more than once within a month and crawling those pages more frequently. Furthermore, to fix problems and complaints that may arise from inconsistencies between the search engine results and the actual Web, Google provides a cached copy of crawled pages. Such inconsistencies may happen due to changes that take place in the content of the crawled URLs. It turns out that a through understanding of the rate of change in Web pages is necessary for understanding the required crawling frequencies for Web pages. Cho and Garcia-Molina have published an excellent study regarding the subject [1]. In their study, they crawled the same 720.000 pages, once a day, for a four month period. A page whose MD5 checksum changed in consecutive crawls was considered as changed. Their experimental results show that 23% of the pages in the .com domain changed daily and 40% of all the pages in their set changed within a week. An expanded version of this study in terms of coverage and in terms of sensitivity to change has been provided by Fetterly et al. [2]. In their study, Fetterly et al. crawled a set of 151 million pages, once a week, for a three month period. Their experimental results reveal that there is a strong relationship between the top-level domain and the frequency of change of a Web page. They have also shown that the greater the document size, the greater the change rate and change frequency of a page.

Both of the studies described above prove that the Web has an enormous change rate. According to Cho and Garcia-Molina's results [1], a crawler, completing its crawling cycle within a month will be missing 50% of the change taking place within the Web. This tells us that the current Web dynamics enforce high refresh rates on any general purpose crawler, which tries to catch Web page changes promptly. Some of the results provided by the above studies may prove to be valuable in determining a refresh policy for a crawler which tries to maximize the freshness of its crawled collection. However, the determination of crawling strategies for high freshness values is still an open question.

- **Coverage and seed selection:** The *coverage* of a crawler can be formalized as the division of the number of pages crawled by the crawler in a crawling session, to the total number of *crawlable* pages. A *crawlable* page is one which can be reached by a crawler and is allowed for crawling. Achieving a high coverage value is pretty much related with seed selection and the amount of resources available to the crawling system. The size of the crawled collection is a field on which current search engine battles are done today. Crawling the whole Web requires a significant amount of storage and time. Even with the enormous resources of state-of-the-art search engines, it turns to be almost infeasible to crawl the whole Web content. Thus, instead of crawling the whole Web, search engines try to crawl somewhat "high-quality" pages and try to keep their collections "fresh". Even with such approaches, the selection of "good" seed pages is still important. Starting from highly connected pages such as *yahoo.com* or *dmoz.org* does not provide efficient results because of the graph structure of the Web. Broder et al. show that the structure of the Web [17] is not fully connected. Finding “good” seed pages and achieving high coverage values in a reasonable amount of time are among important challenges for crawlers.

- **Quality:** Due to the enormous size of the Web and the limitations on the number of pages to be downloaded, many times, it is desirable for crawlers to crawl pages that are more “interesting” and “important” earlier so that, whenever they finish crawling, they will not be missing these high-quality pages in their collection. In order to retrieve high-quality pages, a crawler can modify the order of the URLs that it will be crawling such that more important pages are crawled first. Cho et al. [7] show that ordering the discovered URLs according to their PageRank values, which are calculated from the collected/downloaded collection, provides good collections whenever pages with high overall PageRank or backlink counts are desirable. If we assume that Web pages are nodes of a graph and the hyperlinks among pages are directed edges of that graph, then crawling becomes equivalent to traversing this Web graph. Najork and Wiener [9] show that traversing the Web graph in breadth-first search order yields high-quality collections as well.

- **Hidden-Web:** The state-of-the-art search engines generally crawl from the so-called *publicly indexable Web.* Publicly indexable Web refers to the set of Web pages which can be downloaded by just following the hyperlink structure within the Web. Other than publicly indexable pages, Web has a *hidden* face well kept behind forms, searchable electronic databases, and authorization/registration routines. These dynamic or registration-dependent pages constitute the *hidden-Web.* Raghavan and Garcia-Molina [10] analyze the challenges of hidden Web crawling and propose a design for a crawler that can crawl the hidden Web. Their crawler is a *task-specific*, *human-assisted* crawler. Task-specifity means that they try to crawl specific, predefined topics, and human-assistance means that the crawler is supported with a set of important information related with the crawling topic provided by a human expert. Unfortunately, apart from Raghavan and Garcia-Molina's study, there is not much research published on discovery of the hidden Web. Crawling the hidden Web is another open problem in crawling and stands as a challenge in crawler design.

- **Spider traps:** Unfortunately, not all Web crawlers are designed for the benefit of the Internet community. There are many crawlers developed with the idea of collecting e-mails from pages, or rather simply producing an excessive load on the Internet. The e-mails collected by these bots are generally abused by commercial and spam mails. Many people are aware and annoyed of such bots and some of them choose to simply fight against their activities. The software and sites prepared by these people in order to abuse Web crawlers are called *spider traps*. There are different kinds of spider traps in the Internet [11, 12, 13, 14, 15]. A classic example is *http://spiders.must.die.net*, which is a site that generates infinite number of pages, dynamically, whenever a link within the site is followed. Such a design would trap any crawler recursively crawling this site. An efficient general purpose crawler must find a way to avoid spider traps.

2.3 Structure of *ABC*

In order to validate usefulness of the models that we propose and compare them with the currently deployed models, we decided to implement an efficient parallel crawler. Our crawler, *ABC*, is being implemented in C programming language using MPI libraries. Throughout this section we will try to give a detailed picture of the architecture of an *ABC* crawling agent running on a single host of our parallel system.

ABC agents use synchronous I/O and make use of threading in order to be able to utilize resources such as bandwidth, CPU, and disk. All intermediate data structures are stored within dynamic trie data structures [39], a space utilizing data structure, and kept within memory.

There are different types of threads for accomplishing domain name resolution and page download as well as avoiding server overloads and providing politeness. These threads take their input and write their output to special safe queues which provide mutual exclusion and high performance.

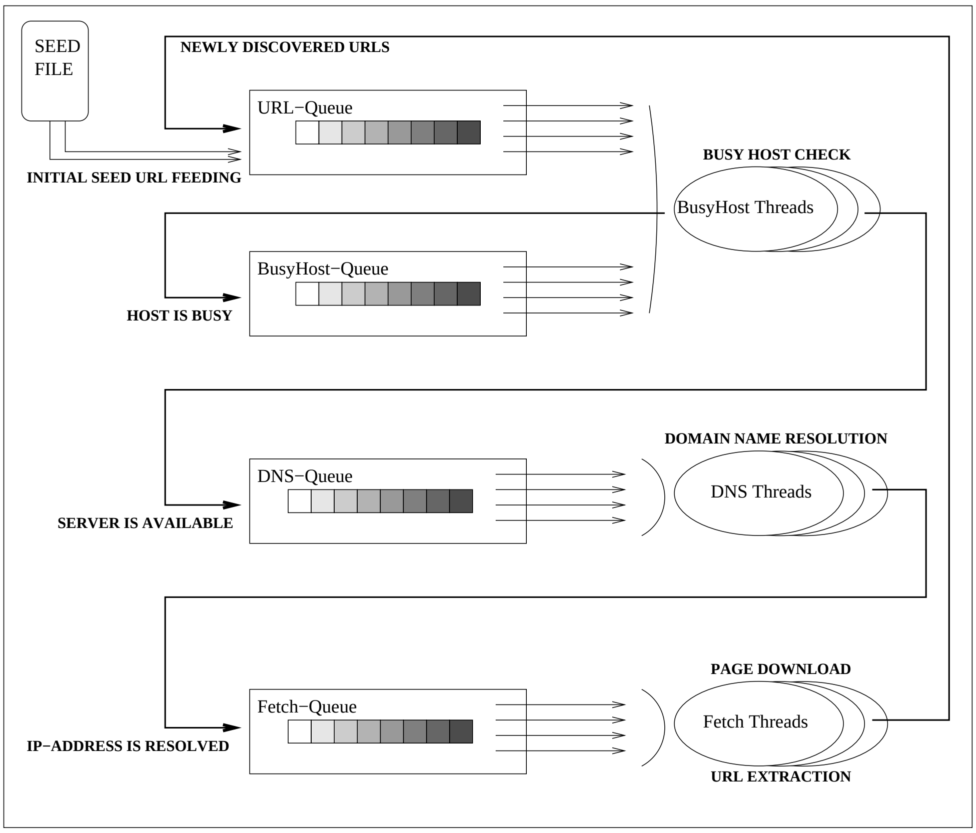


Figure 2.1: Queues and threads of *ABC*

Figure 2.1 shows the data flow between the threads of an *ABC* agent. Initially, *URL-Queue* is filled up with the URLs listed within the SEED file. A *BusyHost thread* parses each URL it receives into its host, port, path, and file parts and checks whether the parsed host was visited within a user-defined amount of time. If so, BusyHost thread sends that URL to the *BusyHost-queue.* If the identified host was not visited within a certain amount of time, the thread puts that URL into the *DNS-queue. BusyHost threads* take their input from the *URL-queue* and the *BusyHost-queue* in a round-robin fashion.

As illustrated in Figure 2.2, *DNS threads* take their input from the *DNS-queue*. Upon receipt of a URL, they check whether the given URL's IP was resolved previously. If so, they directly put the IP address to the *Fetch-queue*. If the given URL's IP address was not resolved previously, DNS threads resolve the IP address for the host of the URL, put that IP address to the *Fetch-queue*, and store the host, IP-address pair in the *HostIP dynamic trie*.

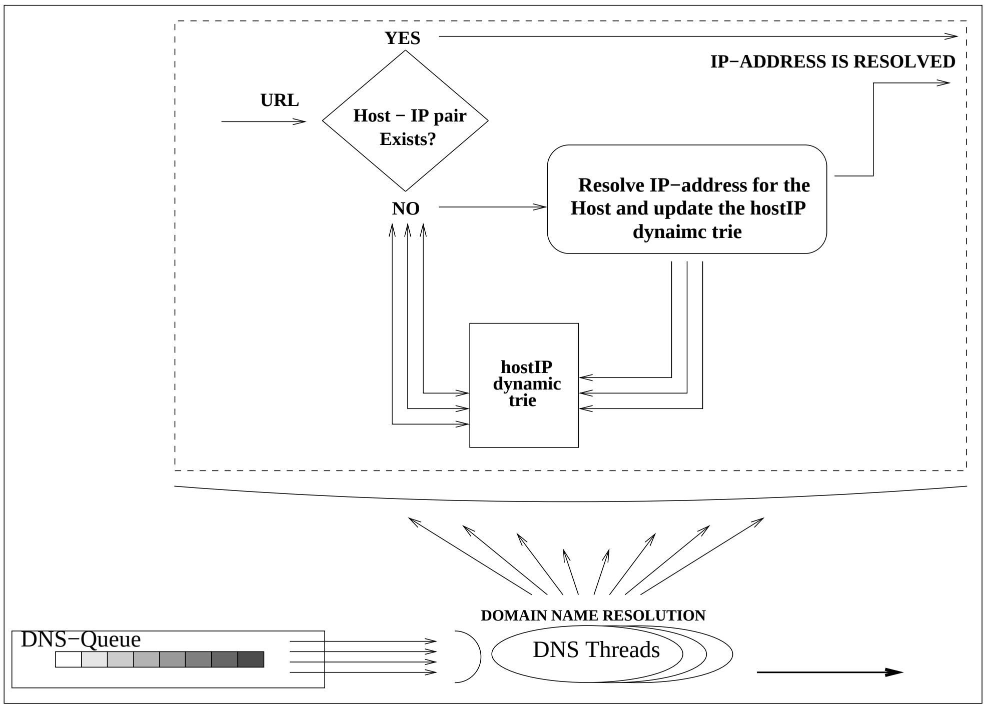


Figure 2.2: DNS threads

In *ABC*, it is the responsibility of the fetch threads to connect to remote servers, download Web pages from those servers, and store the downloaded pages. Before storing downloaded pages, a *contentSeenTest* is done on the downloaded page content. This task is done by taking the MD5 hash value of the content of the downloaded Web page and querying this value in the *contentHash* dynamic trie. If the crawled page's content has not been downloaded/seen before, the content of the page is stored into the *PageRepository*, and its hash value is stored in the *contentHash* dynamic trie. If the page content was crawled before (with some other URL), the page is simply discarded. After saving a page into the

WebRepository, fetch threads extract the links within the downloaded page. Each extracted link goes through a *URLSeenTest*. This is done by querying extracted URLs within the *seenURLs* dynamic trie. As a last step, the *robots.txt* file in the URL server is checked to see whether newly discovered URLs are allowed for crawling. URLs passing this final test are then sent to the *URL-Queue* and added to the *seenURLs* dynamic trie. Already discovered URLs and URLs that point to files which are not allowed by the *Robots.txt* file are simply discarded. This mechanism is illustrated in Figure 2.3.

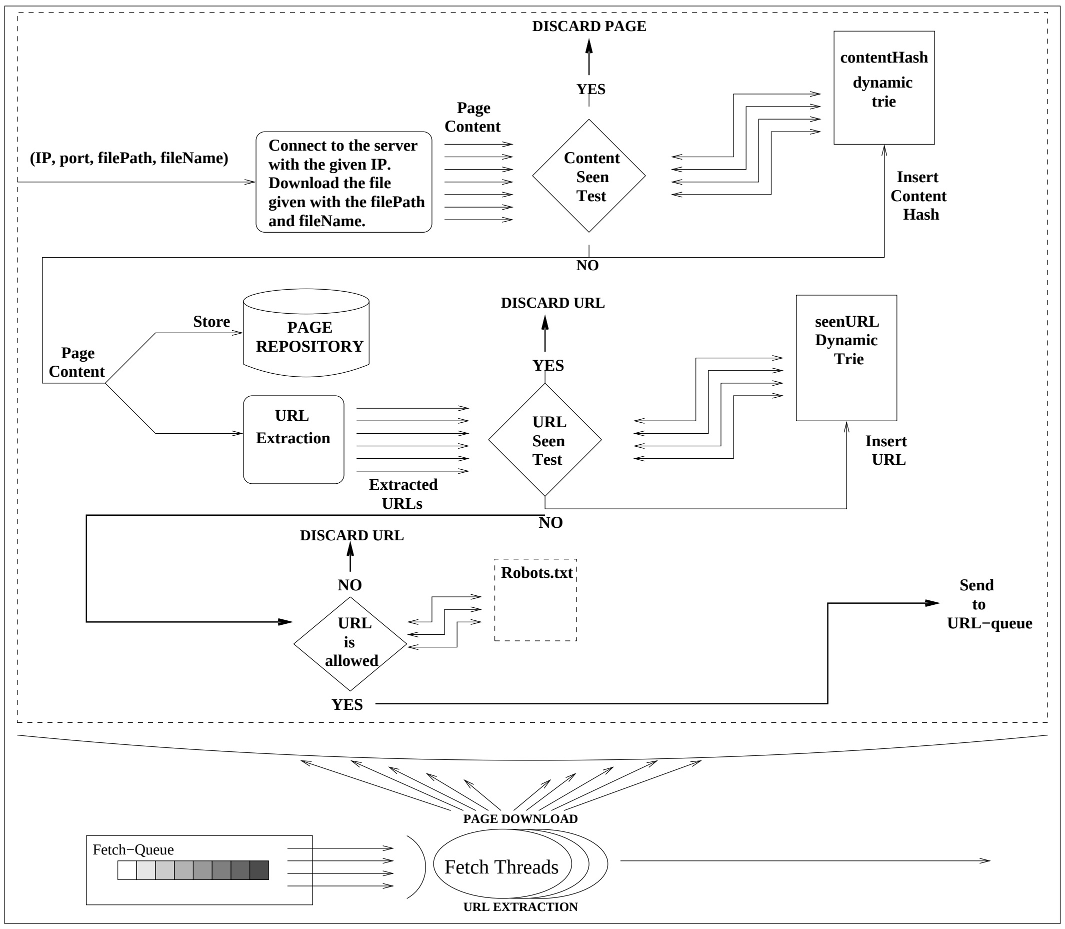


Figure 2.3: Fetch threads

Chapter 3

Parallel Crawlers

The amount of information presented in the World Wide Web and the number of pages providing this information have reached to such a level that it is difficult, if not impossible, to crawl the entire web by a single processor. Thus, current search engines use multiple parallel processors to crawl the Web content. However, due to the competitive nature of the search engine industry, little has been presented to the public about internal structures and design considerations of these engines. Nevertheless, in order to design an efficient parallel crawler, major techniques applied in parallel crawlers and the challenges to be faced must be analyzed. Apart from known and studied problems, we believe that one of the most important problems that have not been studied yet lies in finding efficient page-to-processor assignments. This chapter deals with the parallel crawler architectures and the challenges and advantages of building parallel crawlers. A brief presentation of the *ABC*'s parallel architecture is also provided.

3.1 Introduction

Architectural design choices in crawler design could probably be best identified and presented to public by the engineering teams of the state-of-the-art search engines as they have the chance of observing the practical challenges and have the obligation of producing solutions to these challenges. Unfortunately, providing the expertise and the valuable data gathered through commercial crawling sessions to public use can reduce competitive power of search engines, thus, almost all of the search engines choose not to declare specifications about their crawling techniques and the solutions they develop for the faced challenges. Fortunately, even if computer science technologies are steered by corporate policies depending on profit analysis and user requests/expectations, it is mostly steamed up by academic researches. An excellent example of the meeting of academic knowledge and technological development lies in the history of the foundation of Google, probably the most popular state-of-the-art search engine today. Google has its roots in academical researches. Initiated by Sergey Brin and Lawrance Page throughout their PhD studies, Google was first designed and implemented as a prototype search engine and its structure was presented to the public by Page and Brin [18] in 1998. While explaining their search engine architecture, their paper includes an explanation of the general structure of their crawler and their crawling algorithm as well. Google uses a distributed system of multiple processors for crawling. Unfortunately, focusing on the presentation of a new search engine, Sergey and Brin's paper lacks details about the problems encountered in parallelization of crawling and how did they cope up with those problems.

Unlike the results from commercial researches, the results from academic researches contributing to the field of crawling have steadily increased within the last few years. An extensive study about parallel crawlers was presented by Cho and Garcia-Molina in [8], where they discuss important issues that need to be addressed in parallelization of crawling, advantages of parallel crawlers over single process crawlers, categorization of crawlers according to different aspects, evaluation metrics to evaluate crawler performance, and experimental results gathered from crawls with different crawler architectures. The references [19, 20, 6] all present the design and implementation of distributed/parallel Web crawlers. Major components for parallel crawlers, implementation details, alternatives and design choices are well presented within these references. Boldi et al. [21] presents the implementation of a distributed crawler as well. The assignment function they propose is designed such that it will bring fault tolerance to the system.

3.2 Advantages of Parallel Crawling

Cho and Garcia-Molina discuss the advantages of parallel crawlers over single process crawlers in [8]. We will summarize their observations as we believe that they cover most of the issues valuable enough to be discussed.

- Scalability: Given the current size of the Web, it turns out to be a necessity to use multiple parallel processors to achieve the required download rate, and given the growth rate of the Web, it is easier to scale to the increase of the Web by increasing the number of processors in parallel crawlers, rather than increasing the power/capacity of the hardware components within a sequential system. Parallel crawling architectures are far more scalable than single processor architectures.
- Network-load dispersion: By running each crawler process at geographically distant locations and having them download *geographically-close* pages, the network load can be dispersed over the Internet instead of focusing on a single point.
- Network-load reduction: Distributing crawlers geographically may reduce the network load as well. In such a scheme, crawler processes would be closer to their target pages and this would reduce the network load caused by the crawler as pages will have to go only through the local networks.

In addition to these issues listed by Cho and Garcia-Molina, we believe that reducing the duration the crawling cycle is also an important advantage of parallel crawlers:

- Reduced crawling cycle: A crawling cycle is the time elapsed between the start of a crawling session and the start of the following new crawling session. Generally, within a crawling cycle, crawlers do not download the same pages more than once. An optimistic crawling cycle for crawling the whole Web would take more than a month. Within such a long time period, many of the crawled Web pages become obsolete and thus the Web repository generated loses its value to some extent. Reducing the crawling cycle enhances the freshness of the generated Web repository. Parallel crawlers can reduce the overall crawling time and thus can shorten the crawling cycle significantly as they provide higher download rates and parallel processing of the downloaded data.

3.3 Parallel Crawling Challenges

The crawling problem poses many challenges in its pure self as we have described in Section 2.2. Crawling in parallel adds another level of complexity to this already difficult problem. Generally speaking, main overheads in parallel programs are due to:

- Computational imbalance,
- Communication overhead,
- Redundant computation.

Minimizing these overheads while sacrificing minimum from performance is the single most important challenge in parallel crawling.

In [8], Cho and Garcia-Molina present the major challenges to be addressed in parallel crawlers. We will summarize their analysis here together with our own observations in order to give an understanding of the challenges of parallel crawling. We categorize these challenges according to the overhead types listed above.

- Computational imbalance: An important source of overhead in parallel programs is due to the imbalances in the loads of the tasks assigned to processors. Due to such imbalances, some processors may spend time being idle while other processors are overloaded.

  - Balancing storage and page download requests: In parallel crawling, it is very important to balance the stored data amount and the number of page download requests. Retrieving page contents from remote hosts and storing them is one of the most time-consuming portions of the crawling process. Balancing the storage among the crawling agents will probably make sure that each agent spends roughly equal times in retrieving and saving page contents. Another time consuming operation is running TCP's three-way handshake protocol and opening sockets to retrieve the contents of pages. Balancing the overall page download request numbers that the crawling agents do will probably balance the time spend during the handshake and socket operations. Balancing the loads on the agents will make sure that they will finish their crawls roughly in the same time and thus, will reduce the overall crawling cycle time.

- Redundant computation: Parallel programs may sometimes do redundant computation in order to simplify program design or to reduce the dependencies and interaction overheads.

  - Overlap avoidance/minimization: It is possible for multiple crawlers running in parallel to crawl the same page more than once. If such overlaps increase in number, a degradation in the overall performance of the crawler will be observed. This type of problems may occur in crawlers that have no-coordination among its crawling agents. In order to avoid overlaps, crawling agents of a crawler have to either communicate the downloaded URLs with each other or obey to a page-to-processor mapping strategy which avoids overlaps.

- Communication overhead: Communication of information between the parallel processors is one of the major overheads in execution of parallel programs. Thus, in message passing environments such as MPI, the performance of the system is often measured in time units in order to understand the overhead induced by the communication. The message passing time t_{comm} is usually a linear function of message size and is represented as: $t_{comm} = t_s + mt_t$, where t_s represents the startup time, the time required to handle a message at the sending and the receiving nodes, m represents the size of the data being transferred and t_t represents the transfer time per data unit, a metric related with the bandwidth and includes network as well as buffering overheads. We can figure out that the amount of data communicated and the number of messages sent during communication are both important metrics in determining the overall communication overhead from the formula for t_{comm}. The total volume of the communication messages (MV) and the total number of messages (NM) are two loose upper bounds for the costs induced by communication. In parallel crawling, there are many challenges which can only be solved through communication. This communication may be effective on the crawlers performance if the number of parallel processors increase or the processors are located at geographically distant locations.

– Maximization of coverage: In a parallel crawler where processors only crawl the pages that are assigned to themselves, a problem arises when a crawling agent discovers URLs to pages which are not assigned to itself. If the agent discards those URLs, the coverage of the crawler decreases. In order to maximize the coverage, each crawling agent has to send the URLs that it had discovered to the responsible agents.

– Early retrieval of high-quality pages: Even the state-of-the-art search engines cannot manage to crawl and index the whole content of the Web in required refresh rates. Thus, it is often desirable to increase the quality of the crawled portion of the web. Techniques used in increasing the crawling quality generally use the information extracted from the downloaded portion of the Web. In a parallel crawler, each process has the information of the structure of the Web portion that is assigned to itself. Thus, unless it has the overall crawling information provided by the other crawling processes, a process of a parallel crawler may not be able to make as good crawling decisions as a centralized crawler makes.

3.4 A Taxonomy of Parallel Crawlers

We believe that it is appropriate to categorize parallel crawler architectures depending on the parallel algorithm models. Furthermore, it is known that parallelization of any problem has two major steps. *"Dividing a computation into smaller computations and assigning them to different processors for parallel execution."* [30]. The data decomposition and mapping greatly effects the communication and coordination of the processors. Specifying the type of coordination taking place and the data decomposition and mapping applied is also important in the understanding of the nature of a parallel algorithm, so we will try to elaborate on these issues as well while inducing a categorization depending on the parallelization models. For a detailed explanation of the principles of parallel algorithm design, the reader may refer to [30].

In parallel crawling, there are two major algorithmic models applied: *master-slave* and *data-parallel.*

- master-slave parallel crawling model: In the master-slave parallel crawling model, each processor sends its links, extracted from the pages it downloaded, to a central coordinator. This coordinator, then assigns the collected URLs to the crawling processors. An implicit data partitioning on both the input and output data is implied by the master processor, and the mapping technique is a centralized dynamic mapping. Dynamic mapping techniques distribute the work among processors during the execution. In the parallel crawling problem, the crawling tasks are mostly generated dynamically (through URL discovery) and the size of the data related with each task (just a URL) is relatively small enough to be moved from a processor to the master and back to another processor. Hence, one might believe that dynamic mapping and the master-slave model are suitable for parallel crawling. The weakness of the master-slave approach is that the coordinating processor may become a bottleneck. Within the crawling problem, the number of crawling tasks are very large and even though the size of a single URL may be small, a large number of messages must be communicated between the slave processors and the master for dynamic mapping of all URLs to the crawling processors. Furthermore, even if the crawling tasks are generated dynamically, an accurate estimation of task sizes and numbers for the forthcoming crawls can be made depending on the previous crawls, a property of the crawling problem which favors static mapping. A figure representing the architecture of master-slave parallel crawling model is given in Figure 3.1.

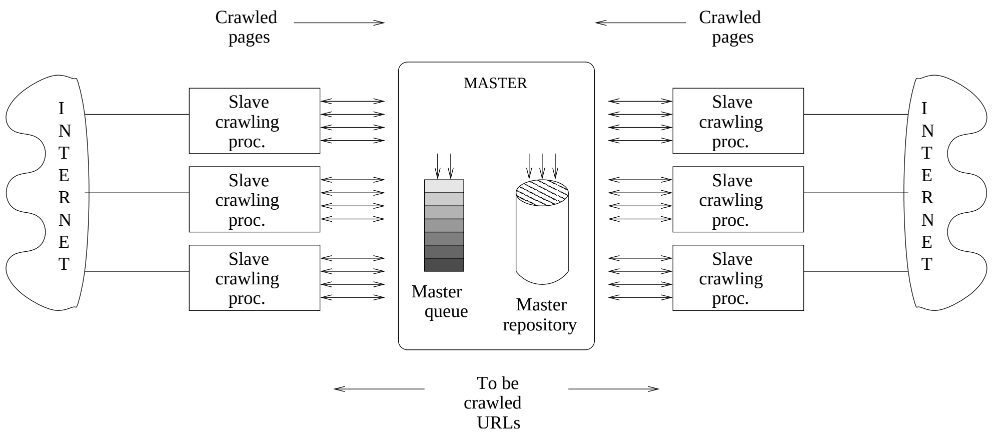


Figure 3.1: Master-slave crawling model

- Data-parallel crawling model: In the data-parallel model, parallelism is achieved by applying the same computation on different data. Depending on the coordination requirements, data-parallel algorithms can be also divided into two: *independent* and *coordinated*.
  - In the *independent data-parallel crawling model*, each processor independently traverses a portion of the Web and downloads a set of pages pointed by the links it had discovered. Since some pages are fetched multiple times in this approach, there is an overlap problem, and hence, both storage space and network bandwidth are wasted. The partitioning scheme adopted is input data partitioning. The input data, namely the seed pages, are partitioned among processors. There is no coordination among the crawling processors and thus there is no need for

communication. The intermediate data and the output data may be replicated unnecessarily. In this approach, data is not mapped to processors. Even though a great amount of data replication can occur, the algorithmic model applied is classified as a data-parallel model due to the initial partitioning of the input data, namely the partitioning of the seed pages.

– In the *coordinated data-parallel crawling model*, pages are partitioned among the processors such that each processor is responsible from fetching a non-overlapping subset of the pages. Since some pages downloaded by a processor may have links to the pages in other processors, these inter-processor links need to be communicated in order to obtain the maximum page coverage and to prevent the overlap of downloaded pages. In this approach, each processor freely exchanges its inter-processor links with the others. Both the input and the output data is partitioned in the data-parallel model and static mapping techniques are applied. There are various static mapping techniques applied in parallel crawling. These techniques may be categorized into two groups: Hash-based and hierarchical mapping techniques. Hash-based techniques are based on the hash value of the URL of a page or the hash of the host part of the URL of a page. Hierarchical mapping techniques use the already existing hierarchy within the URL tree. For example, a processor may crawl pages in the *.com* domain whereas another processor may crawl pages in the *.org* domain. In this study, we propose a new class of mapping techniques based on graph and hypergraph partitioning. A figure illustrating coordinated data-parallel crawling model is given in Figure 3.2.

3.5 Architecture of Parallel Crawlers

In this section, we will try to visualize the architecture and structure of parallel crawlers as well as analyzing the structure of our crawling system *ABC*.

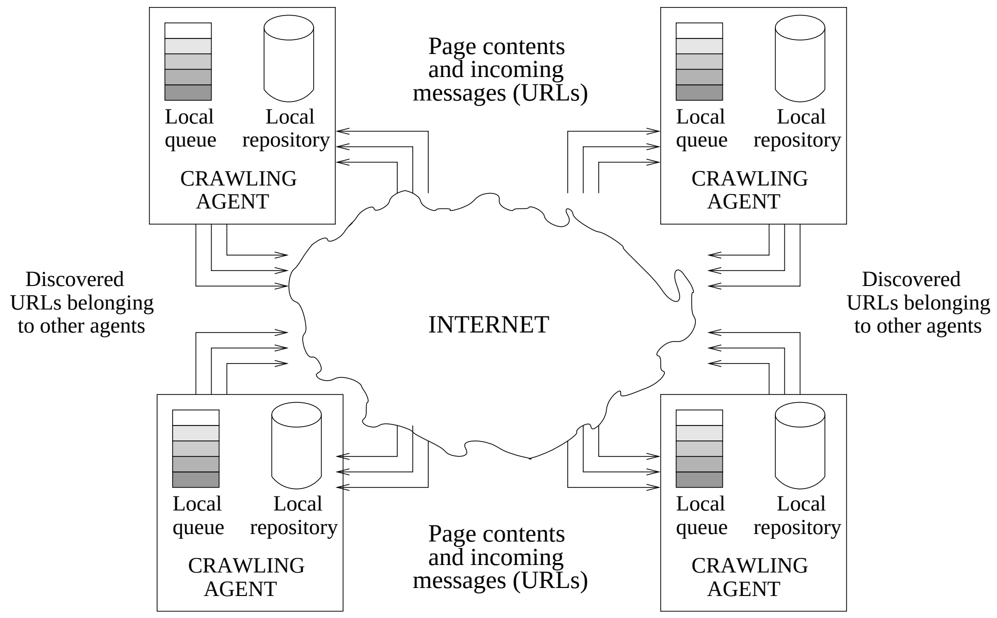

Page contents and incoming messages (URLs)
Local queue
Local repository
CRAWLING AGENT
Discovered URLs belonging to other agents
INTERNET
Page contents and incoming messages (URLs)

Figure 3.2: Coordinated data-parallel crawling model

3.5.1 General Architecture

Within a parallel crawler, there exist multiple crawling processors trying to download and store the Web content in parallel. These crawling processors may be located within the same local Intranet and thus connect to Internet through the same access point, or they may be located at different geographical locations all over the World. A crawler whose processors are located within the same local network may be labeled as an *intranet parallel crawler* whereas a crawler that has distributed its processors at geographically distant locations can be labeled as a *distributed crawler*. When crawling processors are located within the same network, the bandwidth capacity of the local network connection becomes a bottleneck for the overall crawling system. Furthermore, as the network traffic will be focused on a single fixed point within the Internet, it will not be possible to exploit the possibilities for reducing or dispersing the network load. However, the communication between the crawling processors will be faster than a distributed crawler. We illustrate an intranet parallel crawler in Figure 3.3.

A distributed crawler can exploit localism in order to reduce and disperse the

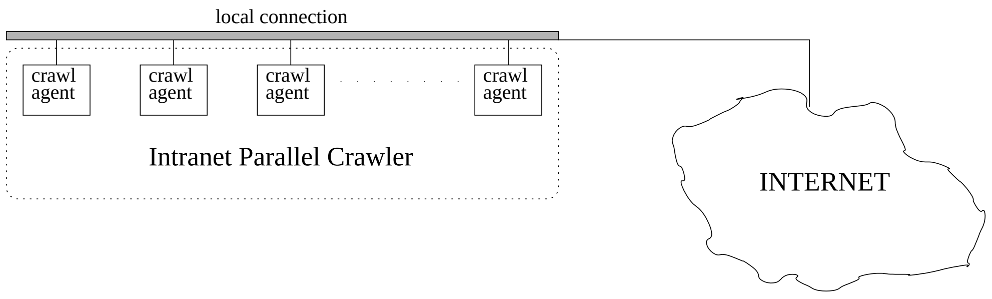


Figure 3.3: Architecture of an intranet crawler

overall crawling network consumption. A distributed crawler whose agents are distributed all over the world is illustrated in Figure 3.4. As it can be understood from the figure, agents of a distributed crawler will often need to communicate through satalite or WAN connections. Hence, communication among the crawling processors will be very slow and costly. Thus, especially in distributed parallel crawlers, analyzing the possibilities for communication reduction is much more valuable.

3.5.2 Architecture of *ABC*

ABC is a coordinated data-parallel crawler, whose crawling processors are running in the same local intranet. Each crawling processor may run several threads to fetch data from multiple servers simultaneously. The crawling space is partitioned among the crawling processors in a non-overlapping fashion. A major assumption in our models is that the crawling system runs in sessions. Within a session, if a page is downloaded, it is not downloaded again, that is, each page can be downloaded just once in a session. The crawling system, after downloading enough number of pages, decides to start another download session and recrawls the Web. For efficient crawling, our models utilize the information (i.e., the Web graph) obtained in the previous crawling session and provide a better page-to-processor mapping for the following crawling session. We assume that between two consecutive sessions, there are no drastic changes in the Web graph (in terms of page sizes and the topology of the links).

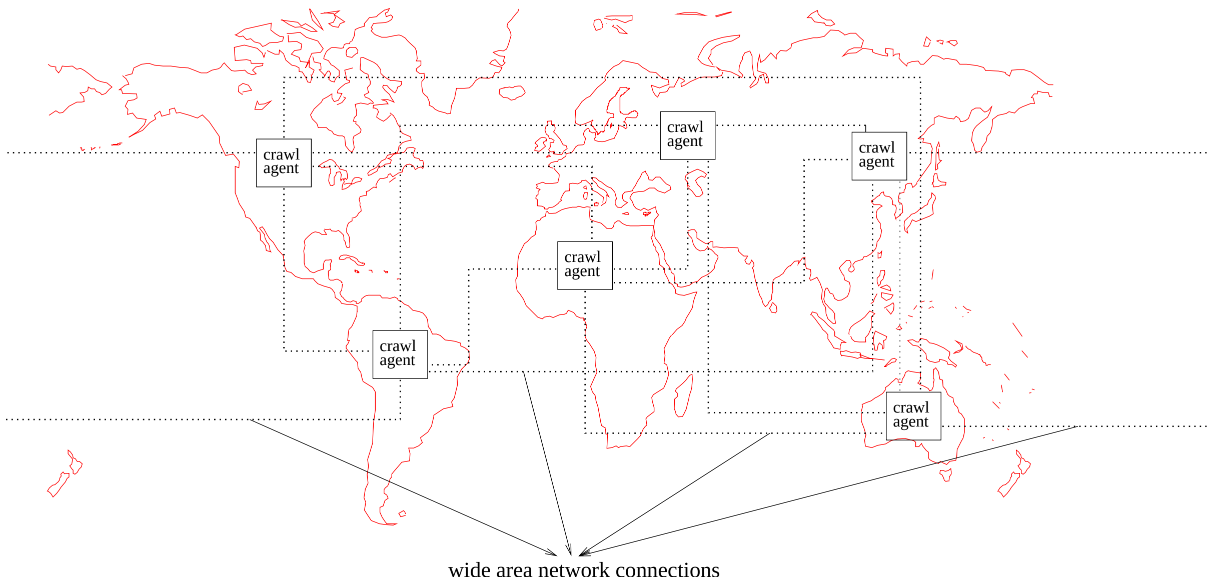


Figure 3.4: Distributed crawler

Whenever a URL belonging to another processor's part is discovered by a processor, the URL is sent to its owner, which induces a communication and coordination among crawling processors. Each processor has a *to-be-crawled queue* and a *downloaded-content repository* of its own. The collected data is stored in a distributed way on the nodes of the parallel system and is used for generating the page-to-processor mapping that will be used on the next crawling cycle. The page-to-processor assignment is determined using our page- or site-based partitioning models prior to the crawling process. The resulting part vectors are replicated at each processor. Whenever a URL which has not been crawled yet is discovered, the processor responsible from the URL is located by the part vector. If a newly found URL which is not listed in the part vector is discovered, the discovering processor becomes responsible from crawling that URL.

Figure 3.5 illustrates *ABC*'s architecture. Our crawling system is being developed on a 24 machine PC-cluster and resides in a local area network with 100Mbps bandwidth connectivity. Unfortunately, nodes of our system do not have network access and thus, running our parallel crawler is infeasible. We are currently in the process of building a new parallel crawling system composing of

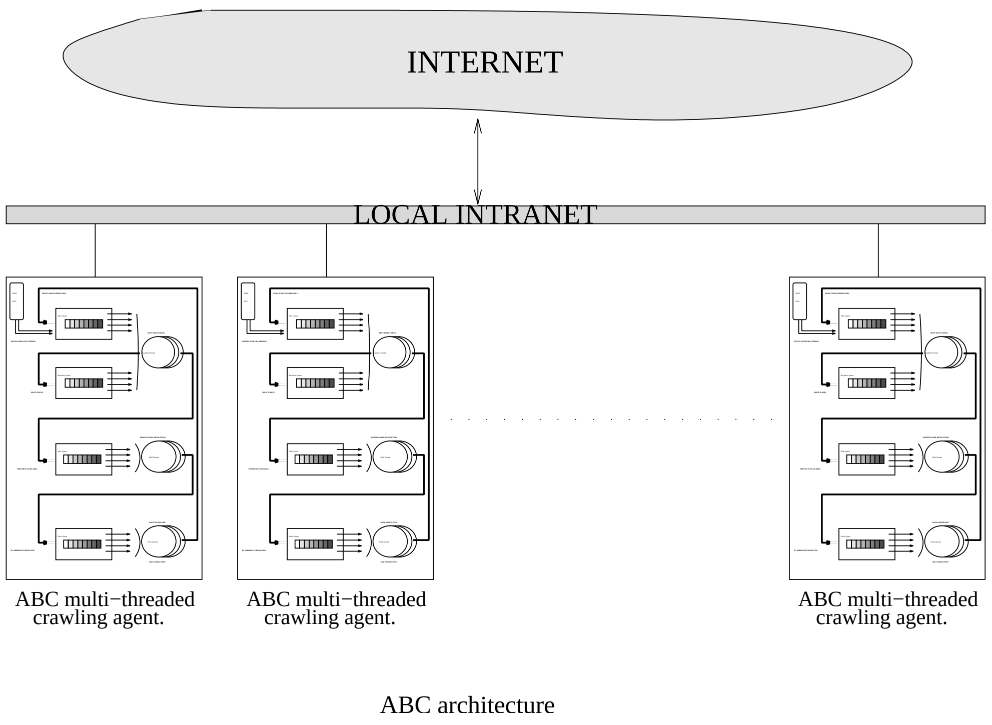

INTERNET
LOCAL INTRANET
ABC multi−threaded crawling agent.
ABC multi−threaded crawling agent.
ABC multi−threaded crawling agent.
ABC architecture

Figure 3.5: Architecture of the *ABC* crawler

48 Intel P4 2.8GHz PC's with 1MB cache, 1GB memory and capable of gigabit network connectivity. We are planning to embed our parallel crawler to this new PC cluster and start testing our crawler by running actual crawls.

Chapter 4

Graph-Partitioning-Based Page Assignment

In general, while parallelizing a serial problem, classical issues such as load balancing and reduction of the communication overhead must be analyzed in depth. Within current parallel crawling designs, load-balancing issues are implicitly solved through page-to-processor assignment functions, which are in fact partitioning functions for parallel crawling. However, to the best of our knowledge, even though they provide a rough load-balancing, existing page-to-processor functions have no effect in reducing the communication overhead. The importance of the communication overhead has been previously observed in parallel crawling community and there is a proposed solution that may reduce the communication overhead through batch communication of messages [8]. However, batch communication requires delaying of messages and trade the crawling quality for minimized communication overhead. Furthermore, the proposed batch communication solution is more like a programming improvement instead of an algorithmic improvement. In this chapter, we propose two graph-partitioning-based page-to-processor assignment algorithms, which minimize the communication overhead significantly. If desired, batch communication of messages can still be applied on top of our algorithms to further reduce the communication overhead. In addition to reduced communication, our algorithms balance both the storage requirements of crawling processors and the number of page download requests issued at each processor concurrently.

4.1 Introduction

As we have stated before in Chapter 3, most of the challenges that are faced in parallel crawling can be solved through communication. The amount of communication required in a crawling session can be determinant on the performance of a crawler. Hence, minimization of the communication overhead turns out to be an important requirement in efficient parallel crawler design. The communication requirements of a parallel crawler can be reduced by efficiently partitioning the data to be crawled among processors.

Existing page-to-processor assignment techniques are either hierarchical or hash-based. Hierarchical approach assigns pages to processors according to the domain of URLs. This approach suffers from the imbalance in processor workloads since some domains contain more pages than the others. In the hash-based approach, either single pages or sites as a whole are assigned to the processors. This approach solves the load balancing problem implicitly. However, in this approach, there is a significant communication overhead since inter-processor links, which must be communicated, are not considered while creating the page-to-processor assignment.

Page-to-processor assignment has been addressed differently by a number of authors. Cho and Garcia-Molina [8] used the site-hash-based assignment technique, a technique that uses host addresses of URLs to feed hash functions that determine the assignment and thus, assigns pages of the same site to the same part, with the belief that this technique will implicitly reduce the number of inter-processor links when compared to the page-hash-based assignment technique. Boldi et al. [21] proposed to apply the consistent hashing technique, a method assigning more than one hash values for a site in order to handle failures among the crawling processors. Teng et al. [31] proposed a hierarchical, bin-packing-based page-to-processor assignment approach. In this chapter, we propose two models based on multi-constraint graph partitioning for load-balanced and communication-efficient parallel crawling.

4.2 Graph Partitioning Problem

An undirected graph $\mathcal{G}=(\mathcal{V},\mathcal{E})$ [32] is defined as a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$. Every edge $e_{ij}\in\mathcal{E}$ connects a pair of distinct vertices v_i and v_j. Multiple weights $w_i^1, w_i^2, \ldots, w_i^M$ may be associated with a vertex $v_i \in \mathcal{V}$. A cost c_{ij} is assigned as the cost of an edge $e_{ij}\in\mathcal{E}$.

$\Pi=\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is said to be a K-way partition of $\mathcal{G}$ if each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, parts are pairwise disjoint, and the union of the K parts is equal to $\mathcal{V}$. A partition Π is said to be balanced if each part $\mathcal{V}_k$ satisfies the balance criteria

$$W_k^m \leq W_{\text{avg}}^m(1+\epsilon), \text{for } k=1,2,\ldots,K \text{ and } m=1,2,\ldots,M. \tag{4.1}$$

In Eq. 4.1, each weight W_k^m of a part $\mathcal{V}_k$ is defined as the sum of the weights w_i^m of the vertices in that part. W_{avg}^m is the weight that each part should have in the case of perfect load balancing. ϵ is the maximum imbalance ratio allowed.

In a partition Π of $\mathcal{G}$, an edge is said to be cut if its pair of vertices fall into two different parts and uncut otherwise. The cutsize definition for representing the cost $\chi(\Pi)$ of a partition Π is

$$\chi(\Pi) = \sum_{e_{ij}\in\mathcal{E}} c_{ij} \tag{4.2}$$

After these definitions, the K-way, multi-constraint graph partitioning problem [33, 34] can be stated as the problem of dividing a graph into two or more parts

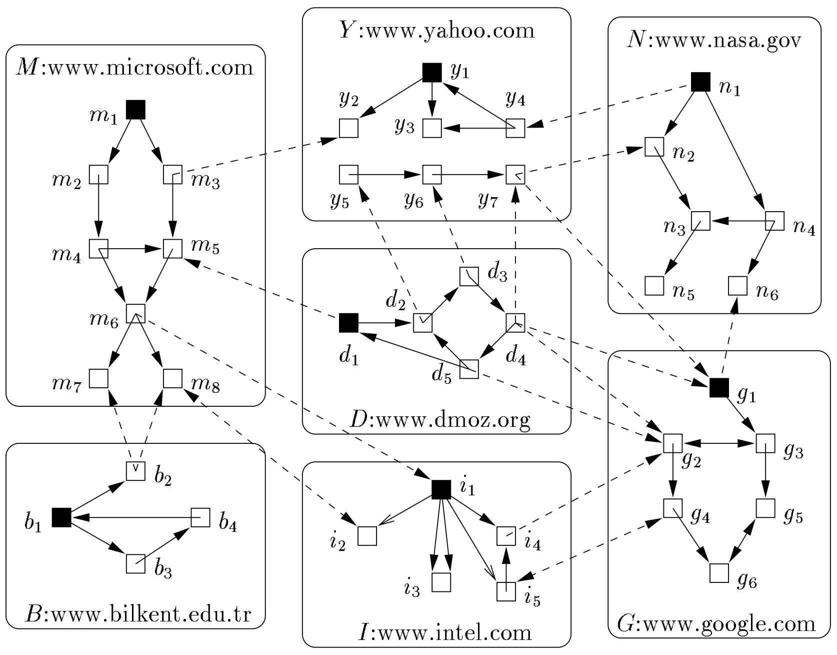


Figure 4.1: An example to the graph structure on the Web.

such that the cutsize is minimized (Eq. 4.2) while the balance criteria (Eq. 4.1) on the part weights is maintained. This problem is known to be NP-hard.

4.3 Page-Based Partitioning Model

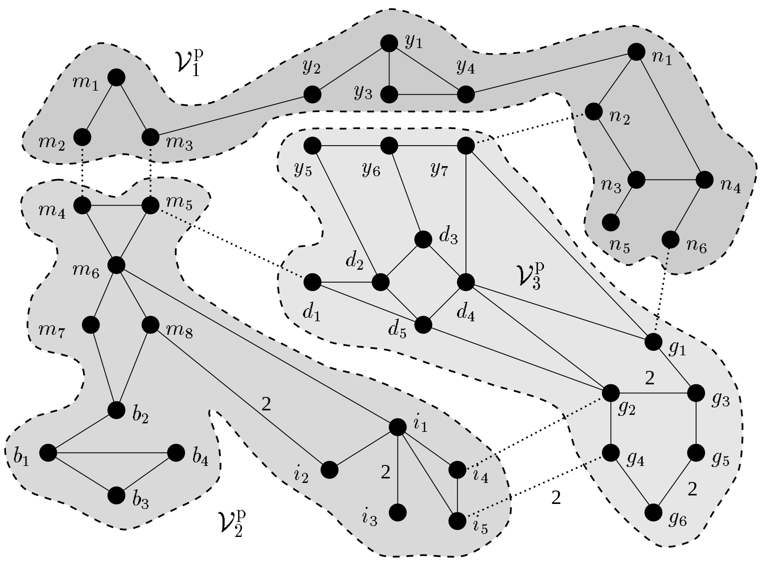
We describe our parallel crawling models on the sample Web graph displayed in Fig. 4.1. In this graph, which is assumed to be created in the previous crawling session, there are 7 sites. Each site contains several pages, which are represented by small squares. The directed lines between the squares represent the hyperlinks between the pages. There may be multi-links (e.g., (i_1, i_3)) and bidirectional links between the pages (e.g., (g_5, g_6)). In the figure, inter-site links are displayed as dashed lines. For simplicity, unit page sizes and URL lengths are assumed.

In our page-based partitioning model, we represent the link structure between the pages by a page graph $\mathcal{G}^{\text{p}} = (\mathcal{V}^{\text{p}}, \mathcal{E}^{\text{p}})$. In this representation, each page p_i

corresponds to a vertex v_i. There exists an undirected edge e_{ij} between vertices v_i and v_j if and only if page p_i has a link to page p_j or vice versa. Multi-links between the pages are collapsed into a single edge. Two weights w_i^1 and w_i^2 are associated with each vertex v_i. The weight w_i^1 of vertex v_i is equal to the size (in bytes) of page p_i, and represents the download and storage overhead for p_i. The weight w_i^2 of vertex v_i is equal to 1, and represents the overhead for requesting p_i. The cost c_{ij} of an edge $e_{ij} \in \mathcal{E}^{\mathrm{p}}$ is equal to the total string length of the links (p_i, p_j) and (p_j, p_i) (if any) between pages p_i and p_j. This cost corresponds to the volume of communication performed for exchanging the links between pages p_i and p_j in case p_i and p_j are mapped to different processors.

In a K-way partition $\Pi^{\mathrm{p}} = (\mathcal{V}_1^{\mathrm{p}}, \mathcal{V}_2^{\mathrm{p}}, \ldots, \mathcal{V}_K^{\mathrm{p}})$ of the page graph $\mathcal{G}^{\mathrm{p}}$, each vertex part $\mathcal{V}_k^{\mathrm{p}}$ corresponds to a subset $\mathcal{P}_k$ of pages to be downloaded by processor P_k. That is, every page $p_i \in \mathcal{P}_k$, represented by a vertex $v_i \in \mathcal{V}_k^{\mathrm{p}}$, is fetched and stored by processor P_k. In this model, maintaining the balance on part weights W_k^1 and W_k^2 (Eq. 4.1) in partitioning the page graph $\mathcal{G}^{\mathrm{p}}$, effectively balances the download and storage overhead of processors as well as the number of page download requests issued by processors. Minimizing the cost $\chi(\Pi^{\mathrm{p}})$ (Eq. 4.2) corresponds to minimizing the total volume of inter-processor communication that will occur during the link exchange between processors.

Fig. 4.2 shows a 3-way partition for the page graph corresponding to the sample Web graph in Fig. 4.1. For simplicity, unit edge costs are not displayed. In this example, almost perfect load balance is obtained since weights (for both weight constraints) of the three vertex parts $\mathcal{V}_1^{\mathrm{p}}$, $\mathcal{V}_2^{\mathrm{p}}$, and $\mathcal{V}_3^{\mathrm{p}}$ are respectively 14, 13, and 14. Hence, according to this partitioning, each processor P_k, which is responsible from downloading all pages $p_i \in \mathcal{P}_k^{\mathrm{p}}$, is expected to fetch and store almost equal amounts of data in the next crawling session. In Fig. 4.2, dotted lines represent the cut edges. These edges correspond to inter-processor links, which must be communicated. In our example, $\chi(\Pi^{\mathrm{p}}) = 8$, and hence, the total volume of link information that must be communicated is 8.



$\mathcal{V}_1^{\mathrm{P}}$
$\mathcal{V}_2^{\mathrm{P}}$
$\mathcal{V}_3^{\mathrm{P}}$
m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8
y_1 y_2 y_3 y_4 y_5 y_6 y_7
n_1 n_2 n_3 n_4 n_5 n_6
d_1 d_2 d_3 d_4 d_5
g_1 g_2 g_3 g_4 g_5 g_6
b_1 b_2 b_3 b_4
i_1 i_2 i_3 i_4 i_5
2

Figure 4.2: A 3-way partition for the page graph of the sample Web graph in Fig. 4.1.

4.4 Site-Based Partitioning Model

Due to the enormous size of the Web, the constructed page graph may be huge, and hence it may be quite costly to partition it. For efficiency purposes, we also propose a site-based partitioning model, which considers sites instead of pages as the atomic tasks for assignment. We represent the link structure between the pages by a site graph $\mathcal{G}^{\mathrm{S}} = (\mathcal{V}^{\mathrm{S}}, \mathcal{E}^{\mathrm{S}})$. All pages belonging to a site S_i are represented by a single vertex $v_i \in \mathcal{V}^{\mathrm{S}}$. The weights w_i^1 and w_i^2 of each vertex v_i are respectively equal to the total size of the pages (in bytes) and the number of pages hosted by site S_i. There is an edge e_{ij} between two vertices v_i and v_j if and only if there is at least one link between any pages $p_x \in S_i$ and $p_y \in S_j$. The cost c_{ij} of an edge $e_{ij} \in \mathcal{E}^{\mathrm{S}}$ is equal to the total string length of all links (p_x, p_y) and (p_y, p_x) between each pair of pages $p_x \in S_i$ and $p_y \in S_j$. All intra-site links, i.e., the links between the pages belonging to the same site, are ignored.

In a K-way partition $\Pi^{\mathrm{S}} = (\mathcal{V}_1^{\mathrm{S}}, \mathcal{V}_2^{\mathrm{S}}, \ldots, \mathcal{V}_K^{\mathrm{S}})$ of graph $\mathcal{G}^{\mathrm{S}}$, each vertex part $\mathcal{V}_k^{\mathrm{S}}$ corresponds to a subset $\mathcal{S}_k$ of sites whose pages are to be downloaded by processor

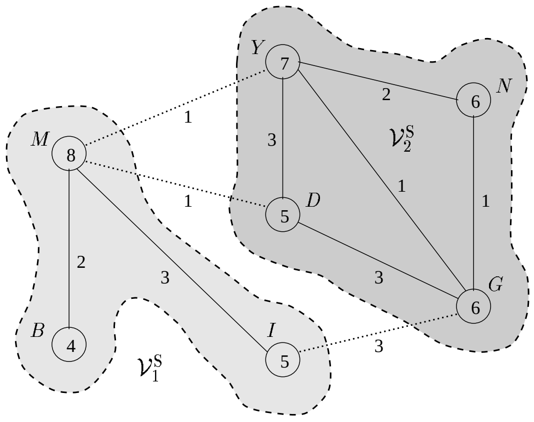


Figure 4.3: A 2-way partition for the site graph of the sample Web graph in Fig. 4.1.

P_k. Balancing the part weights (Eq. 4.1) and minimizing the cost (Eq. 4.2) has the same effects with those in the page-based model.

Fig. 4.3 shows a 2-way partition for the site graph corresponding to the sample Web graph in Fig. 4.1. Vertex weights are displayed inside the circles, which represent the sites. Part weights are $W_1^1 = W_1^2 = 17$ and $W_2^1 = W_2^2 = 24$ for the two parts $\mathcal{V}_1^{\mathrm{S}}$ and $\mathcal{V}_2^{\mathrm{S}}$, respectively. The cut edges are displayed as dotted lines. The cut cost is $\chi(\Pi^{\mathrm{P}}) = 1+1+3 = 5$. Hence, according to this partitioning, the total volume of communication for the next crawling session is expected to be 5.

Chapter 5

Hypergraph-Partitioning-Based Page Assignment

In parallel sparse matrix vector multiplication (SpMxV) problem, usage of graphs to model the communication requirements are pretty common, even though graphs do not truly model the communication volume. In fact, it has been shown [26] that in SPMxV, graphs model a metric which is loosely related with the communication volume. Çatalyürek and Aykanat [24, 27] proposed novel hypergraph models which avoid this deficiency of the graph model. In SPMxV problem, hypergraph models correctly model the volume of communication. On the other hand, in the parallel crawling problem, graph model has no deficiency and correctly models the volume of communication occurring within the parallel system. However, hypergraph models are still valuable for this flavor of problems. For the parallel crawling problem, the hypergraph representation of the Web correctly represents the number of messages that will be communicated among the crawling processes, which is another important metric in minimization of the communication overhead of a parallel system. Even though most of the existing models that try to minimize the communication overhead focuses on minimization of the communication volume believing that minimizing that metric is likely to minimize the overall communication overhead, Uçar and Aykanat [25] show that minimizing the number of communication messages may be as important as minimizing the volume of the communication. Within this chapter, we will provide hypergraph models which correctly model and minimize the number of messages transmitted between the crawling processes.

5.1 Introduction

In order to follow up the proposed models presented in this chapter, it is vital to understand the distinction between minimizing the message volume and minimizing the number of messages. Figure 5.1 is introduced to clarify this distinction. In this figure, we assume that there is a page A which has been assigned to part 0. Page A contains links to other pages and some of the pages pointed by these links are in other parts. Without loss of generality, part i is assumed to be mapped to processor P_i, for $i = 1, 2, ...k$, where k is the number of processors. We see that A has one link to a page in part 0, two links to pages in part 1, three links to pages in part 2, and one link to a page in part 3. Whenever processor P_0, which is responsible from crawling page A, crawls page A and extracts the links within, it will have to communicate the links that belongs to other processors. Actually, processor P_0 will have to send three messages. The messages to be sent to processors P_1, P_2 and P_3 will carry 2, 1, and 3 URLs respectively. The number of messages induced by page A is 3 messages, whereas the communication volume that is induced by page A is $(2 + 3 + 1)$= 6 URLs.

The total communication volume and the total number of messages provide estimations for defining the communication overheads of a parallel program. The communication volume represents the amount of data transfer that will happen between the processors. If we think of the communication formula described in Section 3.3, communication volume can be thought as the sum of message size m's for all of the messages that will be communicated. The number of messages is a self-explanatory term that represents the number of messages that have to be communicated during the execution of the parallel program.

In determining the overall communication overhead, we can use NM and MV

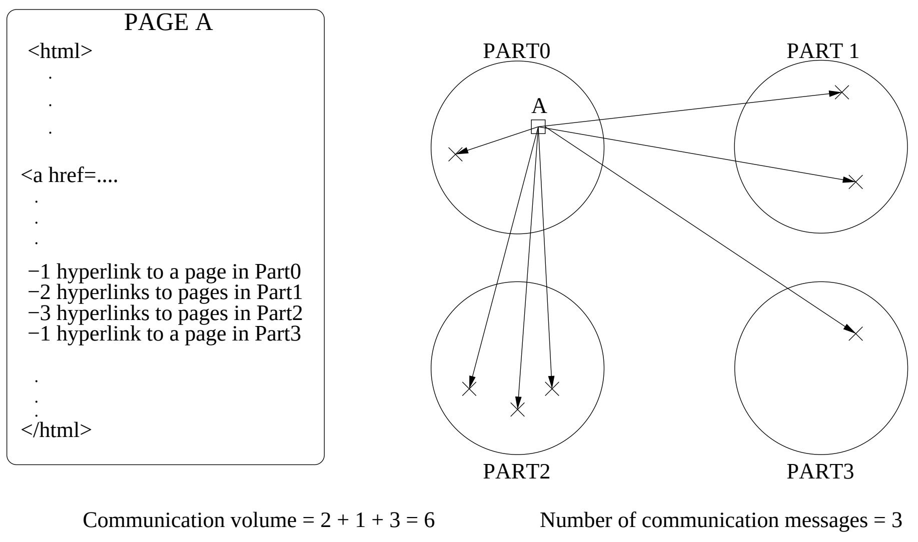

PAGE A
<html>
.
.
.
<a href=....
.
.
.
−1 hyperlink to a page in Part0
−2 hyperlinks to pages in Part1
−3 hyperlinks to pages in Part2
−1 hyperlink to a page in Part3
.
.
.
</html>
PART0
PART 1
A
PART2
PART3
Communication volume = 2 + 1 + 3 = 6
Number of communication messages = 3

Figure 5.1: Communication volume vs. number of messages.

to bring an estimation. The communication volume is multiplied with t_t to give an estimation on the overall transfer overhead, and the number of messages is multiplied with t_s to give an estimation of the overall startup latency. Note that these estimations would give totally exact results if none of the communications were occuring concurrently. Nevertheless, some of the message passing operations are accomplished in parallel. Even so we believe that minimizing NM or MV is a reasonable estimation for the overall communication overhead and reducing them by efficient heuristics is likely to reduce the overall communication overhead. Depending on the problem and the system architecture, the transfer overhead or the startup latency can be the dominant factor within the communication overhead or they may have equal importance. In parallel crawling problem, the messages generally contain URLs which are small in size. A rough average would be 45 bytes for a URL. This implies that, in parallel crawling, reducing the number of messages might be more crucial than reducing the communication volume. Thus, throughout this chapter, we provide two novel hypergraph models which minimize the number of messages during link exchange.

5.2 Hypergraph Partitioning Problem

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$. Every net n_i is a subset of vertices. The vertices of a net are also called its *pins*. The size of a net n_i is equal to the number of its pins, i.e., $|n_i|$. The set of nets that contain vertex v_j is denoted by $Nets(v_j)$. The degree of a vertex v_j is denoted by $d_j = |Nets(v_j)|$. Weights can be associated with vertices.

$\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ is a K-way vertex partition of $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ if each part $\mathcal{V}_k$ is non empty, parts are pairwise disjoint, and the union of parts gives $\mathcal{V}$. In Π, a net is said to *connect* a part if it has at least one pin in that part. The *connectivity set* Λ_i of a net n_i is the set of parts connected by n_i. The *connectivity* $\lambda_i = |\Lambda_i|$ of a net n_i is the number of parts connected by n_i. A net n_i is said to be cut if it connects more than one part (i.e., $\lambda_i > 1$) and uncut otherwise. The set of cut and uncut nets are also referred to as external and internal nets, respectively. In Π, weight of a part is the sum of the weights of vertices in that part.

In hypergraph partitioning problem, the partitioning objective is to minimize the *cutsize*:

$$cutsize(\Pi) = \sum_{n_i \in \mathcal{N}} (\lambda_i - 1). \tag{5.1}$$

This objective function is widely used in VLSI community [22] and in scientific computing community [23, 24, 25] and it is referred to as the *connectivity*-1 cutsize metric. The partitioning constraint is to satisfy a balancing constraint on part weights. A partition Π is said to be balanced if each part $\mathcal{V}_k$ satisfies the balance criteria, i.e.,

$$\frac{W_{max} - W_{avg}}{W_{avg}} \leq \epsilon. \tag{5.2}$$

Here W_{max} is the weight of the part with the maximum weight, W_{avg} is the average part weight, and ϵ is a predetermined imbalance ratio. This problem is NP-hard [22].

A recent variant of the above problem is the multi-constraint hypergraph partitioning [26, 27, 28] in which each vertex has a vector of weights associated with it. In this problem, the partitioning objective is the same as above, however, the partitioning constraint is to satisfy a set of balancing constraints, one for each one of the weights. The balance criteria for this type of problems is:

$$W_k^m \leq W_{\text{avg}}^m(1+\epsilon), \text{for } k=1,2,\ldots,K \text{ and } m=1,2,\ldots,M. \tag{5.3}$$

In Eq. 5.3, each weight W_k^m of a part $\mathcal{V}_k$ is defined as the sum of the weights w_i^m of the vertices in that part. W_{avg}^m is the weight that each part should have in the case of perfect load balancing. ϵ is the maximum imbalance ratio allowed.

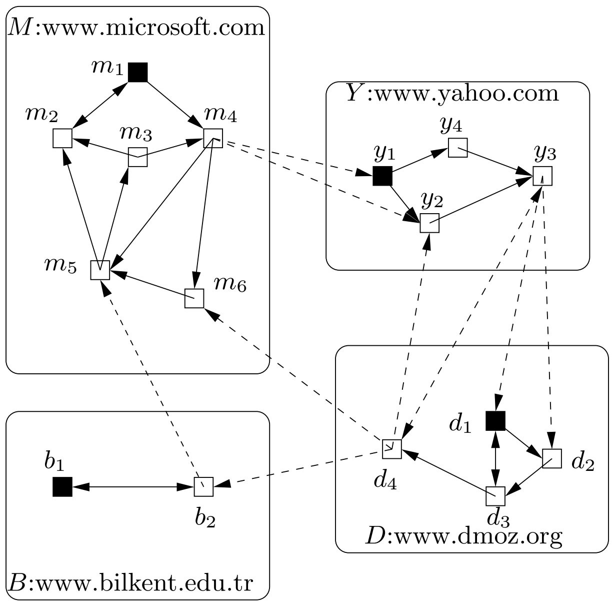


Figure 5.2: Sample Web graph.

5.3 Page-Based Hypergraph Partitioning Model

In our page-based hypergraph partitioning model, we represent the link structure between pages by a page hypergraph $\mathcal{H}^{\text{p}}=(\mathcal{V}^{\text{p}},\mathcal{N}^{\text{p}})$. In this representation, each page p_i corresponds to a vertex v_i. For each vertex v_i, there exists a net n_i whose pins are composed of v_i and the pages that v_i has links to. Vertex v_j is a pin of net n_i either if page p_i has a link to page p_j or $i = j$. Multi-links between the pages are collapsed into a single link between the source page's net and the destination page. Two weights w_i^1 and w_i^2 are associated with each vertex v_i. The weight w_i^1 of vertex v_i is equal to the size (in bytes) of page p_i, and represents the download and storage overhead for p_i. The weight w_i^2 of vertex v_i is equal to 1, and represents the overhead for requesting p_i.

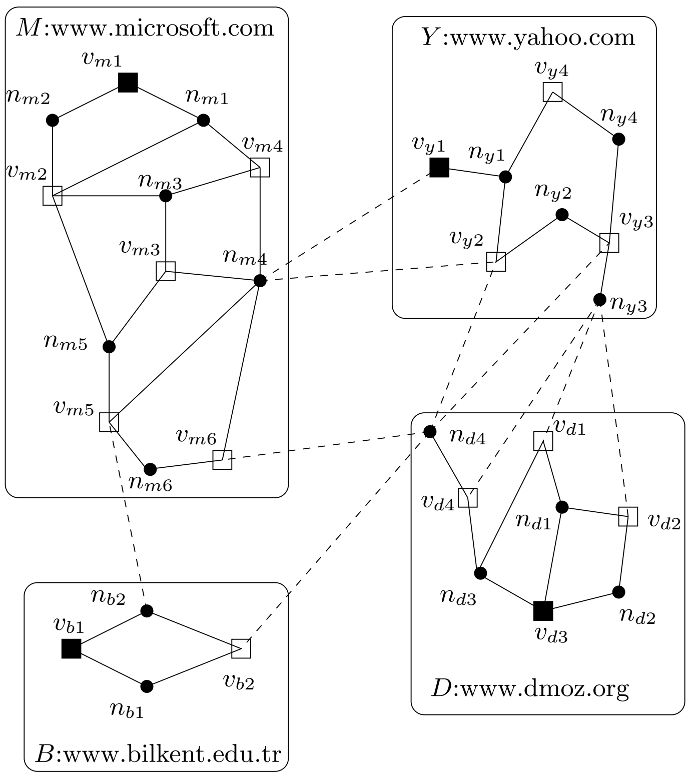


Figure 5.3: Page-based Web hypergraph.

We describe our hypergraph partitioning models on the sample Web graph displayed in Figure 5.2. The corresponding page-based hypergraph is presented in Figure 5.3. In this hypergraph, there are 4 sites. Each site contains several pages, which are represented by small squares. In the figure, inter-site links are displayed as dashed lines between the source page's net and the destination page.

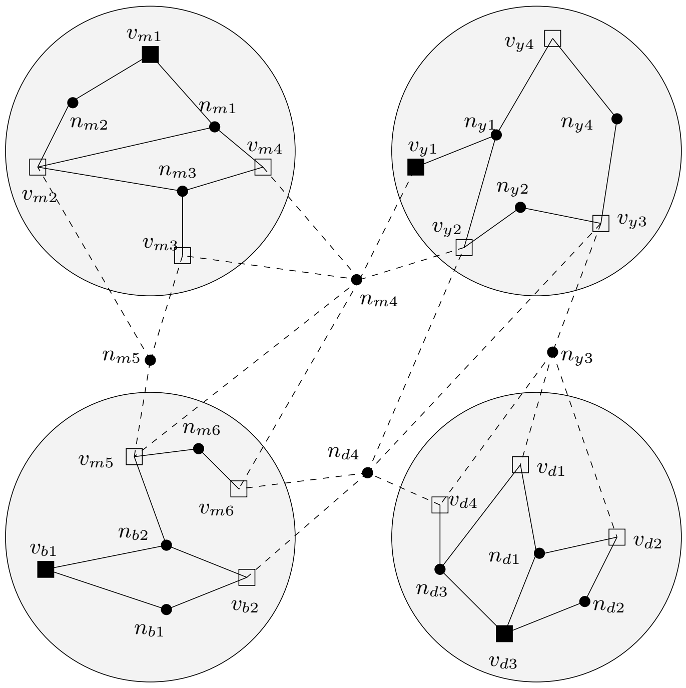


Figure 5.4: A 4-way partition of the page hypergraph in Figure 5.3

In a K-way partition $\Pi^{\mathrm{p}} = (\mathcal{V}_1^{\mathrm{p}}, \mathcal{V}_2^{\mathrm{p}}, \ldots, \mathcal{V}_K^{\mathrm{p}})$ of the page hypergraph $\mathcal{H}^{\mathrm{p}}$, each vertex part $\mathcal{V}_k^{\mathrm{p}}$ corresponds to a subset $\mathcal{P}_k$ of pages to be downloaded by processor P_k. That is, every page $p_i \in \mathcal{P}_k$, represented by a vertex $v_i \in \mathcal{V}_k^{\mathrm{p}}$, is fetched and stored by processor P_k. In this model, maintaining the balance on part weights W_k^1 and W_k^2 (Eq. 5.3) in partitioning the page hypergraph $\mathcal{H}^{\mathrm{p}}$, effectively balances the download and storage overhead of processors as well as the number of page download requests issued by processors. Minimizing the $cutsize(\Pi)$ (Eq. 5.1) corresponds to minimizing the total number of messages during the link exchange between processors.

Figure 5.4 shows a 4-way partition for the page hypergraph in Figure 5.3. For simplicity, unit vertex weights are assumed. In this example, perfect load balance is obtained since weights (for both weight constraints) of the four vertex parts $\mathcal{V}_1^{\mathrm{p}}$, $\mathcal{V}_2^{\mathrm{p}}$, $\mathcal{V}_3^{\mathrm{p}}$, and $\mathcal{V}_4^{\mathrm{p}}$ are four. Hence, according to this partitioning, each processor P_k, which is responsible from downloading all pages $p_i \in \mathcal{P}_k^{\mathrm{p}}$, is expected to fetch and store almost equal amounts of data in the next crawling session. In Fig. 5.4, cut-nets correspond to inter-processor links, which must be communicated. In our example, $cutsize(\Pi) = (\lambda_{n_{m4}}-1)+(\lambda_{n_{m5}}-1)+(\lambda_{n_{y3}}-1)+(\lambda_{n_{d4}}-1) = 2+1+1+2 = 6$, and hence, the total number of messages that must be communicated during the link information exchange is 6.

5.4 Site-Based Hypergraph Partitioning Model

In order to ease the partitioning of the constructed page hypergraph, we also propose a site-based hypergraph partitioning model, which is smaller in size when compared to page-based hypergraph. Another advantage of this model is that it avoids host overloading by assigning all pages of a host to a single crawling agent. Site-based hypergraph model considers sites instead of pages as the atomic tasks for assignment. Even when the atomic tasks are sites, the site-based hypergraph model must have a net for each page in order to be able to correctly measure the number of communication that will occur. Thus, in our site-based hypergraph model, vertices are in site granularity, whereas nets are in page granularity. We represent the link structure between the pages by a site hypergraph $\mathcal{H}^{\mathrm{S}} = (\mathcal{V}^{\mathrm{S}}, \mathcal{N}^{\mathrm{P}})$. All pages belonging to a site S_i are represented by a single vertex $v_i \in \mathcal{V}^{\mathrm{S}}$. For each page p_i, there exists a net n_i. Net n_i has pins on the site that p_i belongs to and the sites of the pages that page p_i has links to. Multi-links between the pages are collapsed into a single edge between the source page's net and the destination site. The weights w_i^1 and w_i^2 of each vertex v_i are respectively equal to the total size of the pages (in bytes) and the number of pages hosted by site S_i. Vertex v_j is a pin of net n_i if there is at least one link between page p_i and any pages in $p_y \in S_j$ or $i = j$. Net costs are equal to one. All intra-site nets, i.e., the nets whose all pins are in the same site, are ignored.

In a K-way partition $\Pi^{\mathrm{S}} = (\mathcal{V}_1^{\mathrm{S}}, \mathcal{V}_2^{\mathrm{S}}, \ldots, \mathcal{V}_K^{\mathrm{S}})$ of hypergraph $\mathcal{H}^{\mathrm{S}}$, each vertex part $\mathcal{V}_k^{\mathrm{S}}$ corresponds to a subset $\mathcal{S}_k$ of sites whose pages are to be downloaded by processor P_k. Balancing the part weights (Eq. 5.3) and minimizing the cost (Eq. 5.1) has the same effects with those in the page-based model.

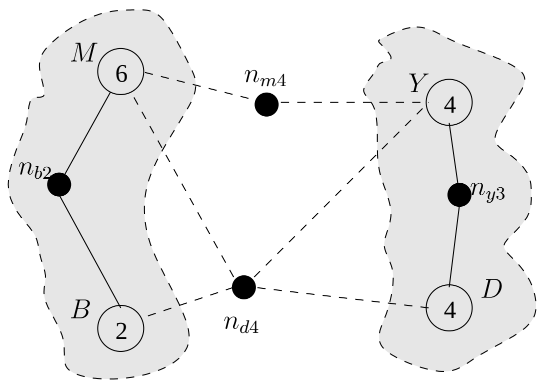


Figure 5.5: A 2-way partition for the site graph of the sample Web graph in Fig. 5.2.

Fig. 5.5 shows a 2-way partition for the site graph corresponding to the sample Web graph in Fig. 5.2. Vertex weights are displayed inside the empty circles, which represent the sites. Nets are represented with small, black, dot-like circles and all net edges have a cost or weight of one. Part weights are $W_1^1 = W_1^2 = 8$ and $W_2^1 = W_2^2 = 8$ for the two parts $\mathcal{V}_1^{\mathrm{S}}$ and $\mathcal{V}_2^{\mathrm{S}}$, respectively. The nets that are in the cut (n_{m4} and n_{d4}) are connected with dotted lines to their pins. The cost is $cutsize = (\lambda_{n_{m4}} - 1) + (\lambda_{n_{d4}} - 1) = 1 + 1 = 2$. Hence, according to this partitioning, the total number of messages that will be communicated during the next crawling session is expected to be 2.

Chapter 6

Experimental Results

Throughout the previous chapters, we have seen the difficulties of crawling and its parallelization, the deficiencies of hash-based techniques in addressing the task assignment in parallel crawling and alternative graph/hypergraph-based models, which can replace hash-based techniques. In this chapter, we will try to evaluate the improvement gained through the usage of the proposed models. A comparison of graph/hypergraph-based techniques and the state-of-the-art hash-based techniques with respect to properties such as the load imbalance in storage or in page request numbers, the communication volume incurred and the number of messages required during the link information exchange will be given.

6.1 Experimental Setup

In this thesis, our main contribution is on the page-to-processor assignment part of the crawling process. To validate the usefulness and applicability of our proposed models, we have implemented a group of programs which take a set of crawled pages in a repository as input, generate the graph or hypergraph structure of this repository from this input and by partitioning this graph or hypergraph, generate an assignment vector for the pages within the repository.

6.1.1 Platform

We have implemented and tested the code that extracts linking structure, generates graph/hypergraph structures, and partitions the generated graph/hypergraph on an Intel Pentium IV 2.54 GHz PC, which has 2 GB of main memory, 512 KB of L2 cache, and 8 KB of L1 cache. All of the code within this set of programs are implemented in C and are compiled in gcc with -O2 optimization option. The state-of-the-art graph partitioning tool MeTiS [33] is used for partitioning the constructed page and site graphs and the state-of-the-art hypergraph partitioning tool PaToH [29] is used for partitioning the constructed page and site hypergraphs. In order to verify the validity of the proposed models, we also developed a data-parallel crawler. Our crawler *ABC* was implemented on a 24 machine Linux PC cluster. Nodes of our cluster are Intel Pentium II 400Mhz machines with 128MB RAM.

6.1.2 Dataset properties

We have tested our code on a crawled set of pages provided by the Google search engine [4]. The dataset is composed of $913,569$ web pages crawled from the US education sites by Google. Pages are collected from $15,819$ sites. The size of this dataset is approximately 8 GB. We have analyzed the properties of the generated graphs and hypergraphs for this dataset and presented them in Table 6.1. According to Table 6.1, average vertex degree for site-based graph is around 2 times larger than that of page-based, which reveals that, even though smaller, the site-based graph is a coarser graph than the page-based graph. The average vertex weight of a site in site graph is 542.71 KB, which is consistent with the values of average vertex weight for a page (9.89 KB) and the average number of pages per site (57.75). Average edge weight for the page graph is given as 45.34 bytes. Remember that edge weights represent the size of the URLs that will be communicated. Thus, we can conclude that on the average, a link URL is incurs 45 bytes of communication overhead. Average edge weight for the site graph is 177.50 bytes. This is due to the fact that some of the links in the page-based

Table 6.1: Dataset properties

Parameters	Page-based	Site-based
Number of pages	913,569	
Number of sites		15,819
Number of vertices	913,569	15,819
Average vertex in-degree	4.90	10.46
Maximum vertex in-degree	5989	999
Average vertex out-degree	4.90	10.46
Maximum vertex out-degree	618	528
Average vertex weight (KB)	9.89	542.71
Maximum vertex weight (KB)	513	90,586
Average number of pages per site		57.75
Maximum number of pages per site		8066
Number of edges	4,480,218	165,450
Average edge weight (byte)	45.34	177.50
Maximum edge weight (byte)	182	401,456
Number of connected components	67,759	243
Maximum sized connected component	805,153	15,576
Number of nets	913,569	913,569
Average net size	5.90	2.79
Maximum net size	619	149
Number of pins	5,393,787	1,350,121
Average vertex degree (hypergraph)	5.90	85.35
Maximum vertex degree (hypergraph)	5990	8855

graph are overlapped as a single edge in the site-based graph. Out of 913,569 pages, 805,153 of them are connected and out of 15,819 sites, 15,576 of them are connected with each other. This tells us that our dataset is a highly connected dataset. Both the average net size and the average vertex degree of the page-based hypergraph are 5.90, basically one more than average out-degree of the vertices of the page-based graph. This is merely due to the fact that there is a net n_i for each page p_i and n_i has a pin to all pages linked by p_i. Thus, each net's size is one more than the out-degree of its originating page. As the size of vertices and nets are equal in our page-based hypergraph, the average vertex degree has to be equal to the average net size. On the other hand, we observe that, for the site-based hypergraph, average vertex degree increases to 85.35 while the average net size decreases to 2.79. This is reasonable since in our site-based hypergraph model, we still have a net for each page and thus, even though the number of vertices decreases to 15,819 from 913,569, the number of nets still remains as 913,569. Basically, a site in the site-based hypergraph is connected to all of the nets that belong to the pages in the site along with the nets that connect to them because of other sites giving link to that site. From Table 6.1 we can see that the site with the maximum number of pages had 8066 pages. This tells us that the cite vertices that has the maximum degree shall have a degree no less than 8066 as it will already be connected to the nets of the pages in itself. We see that the site vertex with the maximum degree has a degree of 8855. The number of pins for the page-based hypergraph is $5,393,787$, basically the sum of the number of edges in the page-based graph ($4,480,218$) and the number of vertices in the page-based graph (913,569), as there is a net for each page which is connected to pages that are linked from that page as well as the page itself. Unfortunately, the number of pins in the site-based hypergraph can not be explained with the parameters in this table. We could have easily guessed that it should be bigger than the number of nets (913,569), but as there are edges that are merged into a single edge in the site-based graph but will be represented with multiple nets in the site-based hypergraph, we can not add the number of edges in the site-based graph (165,450) to the number of nets in the site based-hypergraph to find the number of pins. But we can say that it should be bigger than this sum and the given number of pins in the site-based hypergraph is consistent with this observation.

6.1.3 Experimental parameters

In the experiments, the multi-constraint, multi-level K-way partitioning algorithm of MeTiS and the multi-constraint partitioning algorithm of PaToH are used. The imbalance tolerance is set to 5% for both weight constraints in MeTiS. Due to the randomized nature of the algorithms, experiments are repeated 8 times, and the average values are reported. Results are provided for load imbalance values in storage and page request amounts of processors as well as the total volume of inter-processor communication and the total number of messages occuring in link exchange. The communication volumes presented are given in KiloByte (KB) units. We have conducted a series of experiments to compare the performance of our proposed algorithms with hash-based algorithms. The experiments analyzing the page-based models are conducted on the K values 2, 4, 8, 16, 32, 64, 128, and 256 whereas experiments on site-based techniques are conducted on the K values 2, 4, 8, 16, 32 and 64. This is due to the fact that the number of atomic task units in our site-based graph and hypergraph are very small ($15,819$) and there is no need to partition further when part sizes are very small.

6.2 Experiment Results

Totally, it takes approximately 4 hours to parse the mentioned raw dataset, extract its linking structure, create the graph and/or hypergraph structure and by partitioning the graph and/or hypergraph generate the part vector. Nevertheless, most of this cost is induced by the parsing and graph generation phases which might be done concurrently with crawling. To be more precise, it takes around 3 hours to parse the raw dataset to extract it's page and site-based linking structure. Generation of the graphs and hypergraphs takes about 30 minutes and the time costs for partitioning the graph and hypergraphs of the mentioned dataset is 5 and 10 minutes, respectively. Estimating that a crawling cycle would take a time in the order of weeks, we believe that the pre-processing cost for creating the part vector is negligible.

Table 6.2 displays the imbalance values observed in storage amounts of processors for the page-based assignment techniques. Storage imbalance and number of page download request imbalance values for graph-partitioning-based (GP-based), hypergraph-partitioning-based (HP-based), and hash-based techniques are given. Experiments on page-based assignment show that the hash-based model, our GP-based model, and HP-based model perform roughly equally in balancing both the storage overhead and the number of page download requests.

Table 6.3 shows the imbalance values observed for site-based assignment techniques. In site-based assignment, the GP- and HP-based models outperform the

Table 6.2: Page-based imbalance values

	Page-based					
	Storage Imbalance			Page Request Imbalance		
K	GP-based	HP-based	Hash-based	GP-based	HP-based	Hash-based
2	0.378	0.015	0.237	0.393	0.025	0.433
4	1.231	0.255	0.567	1.271	0.230	0.530
8	3.472	0.414	1.298	3.507	0.446	0.751
16	4.663	0.538	1.751	4.593	0.564	0.900
32	4.708	0.796	2.972	4.715	0.663	1.580
64	4.762	4.091	5.299	4.763	4.448	2.346
128	4.763	7.857	6.982	4.763	4.816	3.006
256	4.765	9.966	8.981	4.777	8.247	5.215

Table 6.3: Site-based imbalance values

	Site-based					
	Storage Imbalance			Page Request Imbalance		
K	GP-based	HP-based	Hash-based	GP-based	HP-based	Hash-based
2	2.910	0.005	9.140	3.133	0.005	2.001
4	3.740	0.313	12.687	3.725	0.331	2.718
8	4.487	3.031	17.137	4.483	3.022	3.101
16	4.749	3.856	18.185	4.724	3.708	3.740
32	4.755	7.499	41.309	4.754	7.735	8.611
64	4.761	11.406	56.777	4.763	10.530	15.442

hash-based approach in balancing the storage requirements as imbalance rates of the hash-based approach deteriorate with increasing K values. This is basically due to the high variation in the sizes of the sites in the dataset used. In balancing the page download request numbers, all three methods perform similarly. Since solution space is more restricted in the site graph and hypergraph, the site-based GP and HP models produce slightly inferior load imbalance rates compared to their page-based counterparts.

Table 6.4 presents the total volume of link information and the total number of messages that must be communicated among the processors for page-based models, whereas Table 6.5 presents the results obtained through site-based techniques. As expected, an increasing trend is observed in communication volumes

Table 6.4: The total volume of communication (in bytes) and the total number of messages during the link exchange for page-based models

	Page-based			
	Message Volume		Number of Messages	
K	GP-based	Hash-based	HP-based	Hash-based
2	1,036,283	101,525,422	55,352	583,151
4	1,678,582	152,337,986	102,085	1,387,089
8	1,962,202	177,796,646	133,260	2,298,673
16	2,273,460	190,489,244	163,817	3,108,535
32	2,445,155	196,846,038	190,847	3,687,317
64	2,971,509	200,033,019	210,704	4,047,882
128	5,237,738	201,627,972	223,107	4,251,617
256	9,090,080	202,417,736	245,029	4,362,796

and message numbers as K increases. According to Table 6.4 and Table 6.5, the proposed GP-based models perform much better in minimizing the total communication volume than hash-based approaches. Site-based hashing results in around 5 times less communication than page-based hashing. This is due to the fact that many inter-processor links are eliminated since sites act as clusters of pages, and almost 4 out of 5 page links turn out to be an intra-processor link when site-based hashing is employed. However, in contrast to the hash-based techniques, the site-based GP model causes an increase in the communication volume. This can be explained by the sparsity of our dataset and the simpler (relative to the page graph) site graph topology which causes a reduction in the solution space. Due to the sparsity of our dataset, there are many pages which do not link each other although they are associated with the same site. By working on the coarser site graph, the MeTiS graph partitioning tool fails to utilize the good edge cuts that cross across the sites (e.g., in Fig. 4.2, pages y_1, y_2, y_3, and y_4 are mapped to $\mathcal{V}_1^{\mathrm{p}}$ while y_5, y_6, and y_7 are mapped to $\mathcal{V}_3^{\mathrm{p}}$). Consequently, the site-based GP model results in partitions with higher cut costs and hence communication volumes.

By analyzing Table 6.4 we can observe that the page-based HP model greatly outperforms hash-based model in reducing the number of messages. However, even though the site-based HP model performs slightly better than the site-based

Table 6.5: The total volume of communication (in bytes) and the total number of messages during the link exchange for site-based models

	Site-based			
	Message Volume		Number of Messages	
K	GP-based	Hash-based	HP-based	Hash-based
2	1,735,560	13,475,898	14,002	10,725
4	2,625,363	19,986,309	23,872	25,681
8	3,152,428	23,046,239	33,687	44,791
16	3,464,325	24,645,513	41,698	67,022
32	4,229,328	25,582,315	51,958	89,547
64	5,723,167	26,205,765	83,590	110,535

hash model, we do not see such a big difference between the site-based HP model and its hash-based counterpart. This is again due to the fact that working on the coarser hypergraph, PaToH fails to utilize good cuts that cross across sites. Consequently, the site-based HP model results in partitions with higher cut costs and hence higher number of message requirements.

We observe from Figure 6.4 that, in order to crawl an 8 GB Web content in page granularity, a crawler with 64 processors will need to communicate 200 MB of data if it uses a hash-based assignment scheme. On the other hand, if the crawler uses our GP-based model, the required volume of communication will be around 3 MB. Again for a 64 processor crawling same dataset requires sending of around 4 million messages in the hash-based approach, whereas our HP-based assignment scheme requires around $210,000$ messages. As illustrated in Figure 6.5, while crawling 8 GB of data in site granularity, the communication volumes generated by hash-based and GP-based schemas for 64 processors are 26.2 MB and $5,5$ MB respectively. The number of messages generated for the same dataset by hash-based and HP-based approaches are $110,535$ and $83,590$ respectively.

We believe that the reduction produced by the GP-based model in the communication volume is very important and through such a reduction, it is obvious that the performance of the crawler would increase significantly. The improvement generated by HP-based models would be more significant and beneficial especially if the crawling agents are located on geographically distributed locations. Note that in a distributed crawling system, the startup time of the communication takes significantly large values and thus, the number of message communications may be the dominant factor in the communication overhead in such systems. We would suggest the usage of GP-based approaches for an intranet crawler and the usage of HP-based approaches for a distributed crawler.

Chapter 7

Conclusion

Throughout this thesis, we mainly focus on designing elegant page assignment models for the parallel Web crawling problem. We summarize what we have discussed throughout the previous chapters here. In Chapter 1, we have given a brief introduction to the parallel crawling and page-to-processor assignment problems together with a guide to the upcoming chapters. An analysis of serial Web crawlers are presented in Chapter 2. We define the basic crawling algorithm, the crawler components, the challenges in Web crawling and the structure of our crawler *ABC*. Chapter 3 is designed in order to give an understanding of parallel crawlers and their implications. We present the necessity of parallelism in Web crawling problem, the advantages of parallel crawlers, the challenges in parallel crawler design, parallel crawler architectures, a categorization of parallel crawlers according to parallelization models and where in this categorization our crawler *ABC* falls in. Having explained the implications, we propose a graph-based assignment technique which greatly reduces the communication volume during link exchange while balancing storage and the number of page download requests in the crawling agents in Chapter 4. We explain two models based on graph partitioning that differ only in their granularity. In the following section we propose another assignment technique which is based on hypergraph partitioning. This assignment technique minimizes the number of messages during the link exchange, thus, reduces the communication overhead while still balancing the storage and page download requests. In Chapter 6 we experiment on our models comparing them with the hashing technique. An explanation of the dataset on which we have conducted our experiments is presented firstly, storage and page download request imbalance values for hash-based, graph-based, and hypergraph-based techniques are given. The total communication volume and the number of messages induced by the proposed techniques are compared with those of hash-based technique and the results are analyzed in depth.

All of our studies depend on the observation that there is room for algorithmic improvements in the state-of-the-art page-to-processor assignment schemas. We proposed two different types of page assignment techniques depending on graph-partitioning and hypergraph-partitioning. Our empirical findings prove that the proposed algorithms can greatly be used to reduce the communication overhead in parallel Web crawling while providing good load balancing on storage or the number of page download requests.

For future research, we consider working on the following issues.

- Actual crawl: We haven't been able to test our proposed models on an actual crawl. Instead of working with pre-crawled datasets, working with data that we have crawled would bring more insight to our analysis about our algorithms. We are currently on the process of establishing a PC cluster on which we can test our models.
- Implementation of a parallel hypergraph partitioning tool: A distributed or parallel crawler generally stores the crawled pages in a local repository. Hence, in a parallel crawler there are multiple repositories that constitute the overall crawled dataset. Partitioning this dataset requires the usage of parallel partitioning tools. For graph partitioning, we have ParMeTiS [35, 36, 37, 38] which can partition graphs in parallel. Unfortunately, PaToH does not have a parallel hypergraph partitioning utility, thus, implementing a parallel hypergraph partitioning tool is necessary to generalize our assignment techniques.
- Hybrid model: In this thesis, we have presented two distinct models, one

minimizing the total volume of communication, other minimizing the number of communication. We are planning to work on a hybrid model which combines the benefits gained from these two approaches.

Bibliography

[1] J. Cho and H. Garcia-Molina. The Evolution of the web and implications for an incremantal crawler. In *Proc. of the 8th Int. World Wide Web Conference (WWW8)*, May 1999.

[2] D. Fetterly, M. Manasse, M. Najork, and J. Wiener. Crawling towards light: A large scale study of the evolution of Web pages. In *First Workshop on Algorithms and Models for the Web-Graph*, Vancouver, Canada, November 2002.

[3] Google Information for Webmastars. http://www.google.com/webmasters/2.html, 2004.

[4] Google Programming Contest http://www.google.com/programming-contest, 2004.

[5] E. T. O'Neill, B. Lavoie, and R. Bennett. Trends in the evolution of the public Web: 1998-2002. D-Lib Magazine, Vol. 9, no: 4, April 2003.

[6] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. In *World Wide Web*, Vol. 2, no: 4, pages 219–229, 1999.

[7] J. Cho, H. Garcia-Molina, and L. Page. Efficient Crawling Through URL Ordering. In *7th Int. World Wide Web Conference*, May 1998.

[8] Junghoo Cho, Hector Garcia-Molina. Parallel crawlers. *World Wide Web 2002*, pages 124–135, 2002.

[9] Marc Najork, Janet L. Wiener. Breadth-first crawling yields high-quality pages. *World Wide Web 2001*, pages 114–118, 2001.

[10] Sriram Raghavan, Hector Garcia-Molina. Crawling the Hidden Web, In *Proceedings of the 27th VLDB Conference*, Roma, Italy, 2001.

[11] Spider traps - an upcoming arms race. http://www.jahns-home.de/rentmei/html/sptraps.html#preamble, 2004.

[12] Web Spider Traps. http://danzcontrib.free.fr/en/pieges.php, 2004.

[13] How to build a Bot Trap and keep bad bots away from a web site. http://www.kloth.net/internet/bottrap.php, 2004.

[14] ROBOTCOP, robots.txt: it's the Law. http://www.robotcop.org, 2004.

[15] The E-Mail Protector Script. http://www.siteware.ch/webresources/scripts/perl/emp.htm 2004.

[16] The robots exclusion protocol. www.robotstxt.org.

[17] A.Z.Broder, S.R.Kumar, F.Maghoul, P.Raghavan, S.Rajagopalan, R.Stata, A.Tomkins, and J.Wiener. Graph structure in the web: experiments and models. In *Proc. 9th WWWConf.*, 1999.

[18] Sergey Brin and Lawrance Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, pages 107-117, April 1998.

[19] Vladislav Shkapenyuk and Torsten Suel. Design and Implementation of a High-Performance Distributed Web Crawler, In *International Conference on Data Engineering*, pages 357-368, 2002.

[20] Demetrios Zeinalipour-Yazti and Marios D. Dikaiakos. Design and Implementation of a Distributed Crawler and Filtering Processor. In *Next Generation Information Technologies and Systems*, pages 58-74, 2002.

[21] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. To appear in *Software: Practice & Experience*, 2004.

[22] T. Lengauer. Combinatorial Algorithms for Integrated Circuit Layout. Wiley–Teubner, Chichester, U.K., 1990.

[23] C. Aykanat, A. Pınar and Ü. V. Çatalyürek. Permuting Sparse Rectangular Matrices into Block-Diagonal Form. In Scicomp, 2003.

[24] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-Partitioning Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. In *IEEE Trans. Parallel and Distributed Systems*, Vol. 10, no:7, 673–693, 1999.

[25] Bora Uçar and Cevdet Aykanat. Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies. In *Scicomp*, 2003.

[26] Ü. V. Çatalyürek. Hypergraph Models for Sparse Matrix Partitioning and Reordering. PhD. thesis submitted to Bilkent University, Computer Engineering and Information Science Dept., Nov, 1999.

[27] Ü. V. Çatalyürek and Cevdet Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. Proceedings of Scientific Computing 2001 (SC2001), p10-16, Denver, Colorado, November, 2001.

[28] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint Hypergraph Partitioning. Tech. Report in University of Minnesota, Department of Computer Science/Army HPC Research Center, no: 99-034, Minneapolis, MN 55455, November, 1998.

[29] Ü. V. Çatalyürek and Cevdet Aykanat. PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey, 1999.

[30] A. Grama, A. Gupta, G. Karypis and V. Kumar. Introduction to Parallel Computing (Second Edition). Addison Wesley, 2003.

[31] S. Teng, Q. Lu, M. Eichstaedt, D. Ford, and T. Lehman. Collaborative Web crawling: Information gathering/processing over Internet. In *32nd Hawaii International Conference on System Sciences*, 1999.

[32] C. Berge. Graphs and hypergraphs. North-Holland Publishing Company, 1973.

[33] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. In *Journal of Parallel and Distributed Computing*, Vol. 48, no: pages 96–129, 1998.

[34] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, pages 296–310, 2000.

[35] Kirk Schloegel, George Karypis, and Vipin Kumar. A Unified Algorithm for Load-balancing Adaptive Scientific Simulations. In *Supercomputing*, 2000.

[36] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *EuroPar*, 2000.

[37] George Karypis and Vipin Kumar. A Coarse-Grain Parallel Formulation of a Multilevel k-way Graph Partitioning Algorithm. In *SIAM*, Philadelphia, 1997.

[38] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. In *JPDC archive*, Vol. 48 , No: 1, pages: 96 - 129, January, 1998.

[39] Alan L. Tharp. File Organization and Processing. John Wiley and Sons, 1988.