A Parallel Crawling Schema Using Dynamic Partition

Shoubin Dong, Xiaofeng Lu, and Ling Zhang

Network Research Center, South China University of Technology,
510640 Guangzhou, China
{sbdong,xflv,ling}@scut.edu.cn
Tel: (8620)87110014
Fax: (8620)87110019

Abstract. Parallel crawling is a key issue for search engine. In this paper we propose a parallel crawling schema based on dynamic partition, in order to fully utilize the available resources and achieve the best of load balance. The crawling schema is evaluated based on parallel metrics and performance of load balance. A prototype system built on Grid middleware has been constructed to demonstrate its efficiency and flexibility.

1 Introduction

As the size of the web is growing explosively, web search engines are becoming increasingly important as the primary means to retrieve information on the Internet. Most search engines use parallel web crawlers to retrieve large collections of web pages, in order to achieve a maximized download rate. However the competition among parallel crawling may result in redundant crawling and wasted resources.

Several researches have been conducted on parallel and distributed crawlers [1-7]. Some features of Google have been introduced in [1], where the crawling mechanism is described as a two-stage procedure. First, a URL server sends URLs to several web crawlers, where pages are fetched in parallel. Second, the downloaded pages are sent to the a central indexer, in which new URLs are parsed out for PageRank computing, and then forwarded to the URL server for next crawling. UbiCrawler[2] is a scalable fully distributed web crawler, in which each agent/robot is responsible for approximately the same number of URLs using Identifier–Seeded Consistent Hashing to achieve load balancing. WebRace[3] is a java-implemented distributed crawler, to collect, annotate and disseminate information from heterogeneous Internet source and protocols. Shkapenyuk and Suel [4] gave a detailed description of the architecture of a distributed crawler. It primarily discusses about the I/O and network efficiency aspects of a crawling system and the scalability issues in terms of crawling speed and number of participating nodes. Hash function is engaged to partition the space of all possible web URLs. Walker [5] used MPI and genetic programming to simulate the results for parallel pseudo-search engine. Travatore [6] is a kind of platform-independent distributed crawler, of which an agent is consisted of three elements: store, frontier and controllers. In [7], Cho proposed a general parallel crawling structure, and evaluated some crawling schema based on static partition according to the parallel metrics.

In our work, we propose a new parallel crawling schema based on dynamic partition mechanism. Based on the dynamic partition, we are able to extend the parallel crawler to run on grid nodes, thus to construct a high performance, fault tolerant and scalable crawler in grid environment.

2 Parallel Crawling Architecture

The parallel crawling architecture, shown in figure 1, is composed of agents and coordinators. An **agent** is a grid node, it performs its download task by running several threads, each of which is called **C-proc**. Each C-proc is dedicated to the visit of a single site. We make sure that different C-proc visit different sites at the same time, so that each site is not overloaded by too many requests. A **coordinator** is to coordinate the behavior of the agents. It may be also an agent.

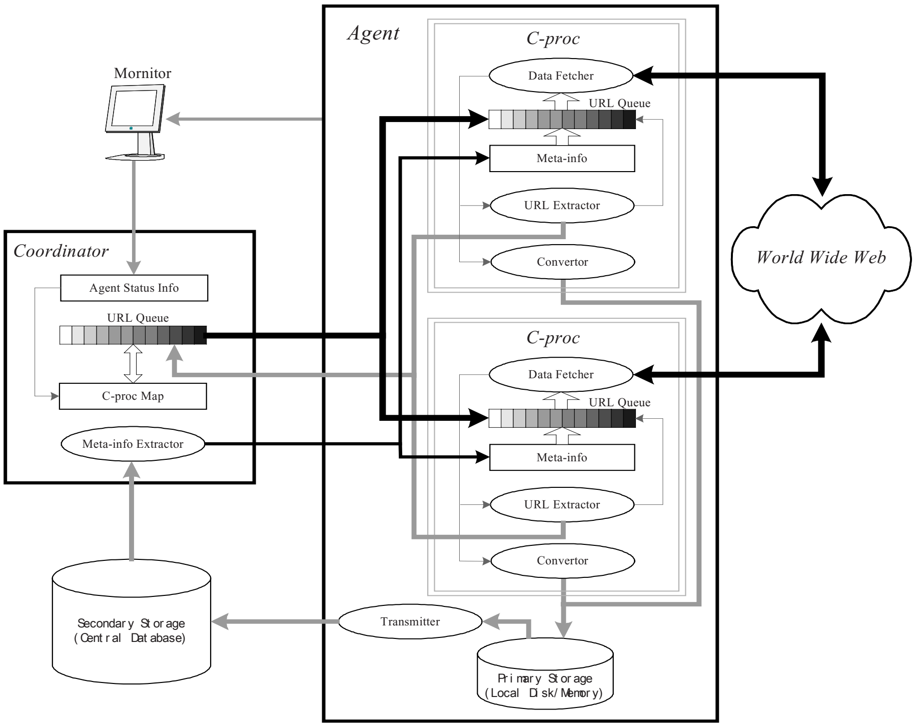

Fig. 1. The Parallel Crawling Architecture

Before the crawler starts the crawling procedure, a number of seed URLs, which are grouped by site names and ordered by prior knowledge, if available, have to be set to the "URL Queue" in the coordinator. The monitor collects the status (such as CPU usage, memory usage, disk size, network bandwidth, etc.) of every agent, and save the information in the "Agent Status Info" of coordinator. When the crawler gets started, the coordinator assigns the number of available C-proc's groups of seed URLs to appropriate C-proc's, based on agent status information and our dynamic partition algorithm, and uses "C-proc Map" to keep the information about which site each C-proc is responsible for at present. Meanwhile, the "Meta-Info Extractor" collects statistics of the formerly downloaded objects in the central database, and distributes them to the corresponding C-proc's, so that C-proc can determine whether pages within the site need to be downloaded. C-proc keeps this statistics in its "Meta-Info".

Objects are retrieved by "Data Fetcher" and then sent to "Convertor", where objects including web pages and objects of other formats are converted into XML formats and saved to the primary storage – agent's local memory or disk, which is supervised by a "Transmitter" located in the same agent. When the number of documents in the primary storage grows to certain degree, all data in the primary storage would be packed and sent to the central database through the transmitter. Meanwhile "URL extractor" extracts new URLs from the downloaded objects and sends those that are local to C-proc's assigned site to the "URL queue" of C-proc, while sends others back to the "URL queue" in the coordinator. Meanwhile "URL extractor" extracts new URLs from the downloaded objects and sends those that are local to C-proc's assigned site to the URL queue of C-proc, while sends others back to the "URL queue" in the coordinator. When some C-proc's URL queue is empty, it asks the coordinator for new assignment. These steps would repeat until there is no URL in the URL-queues of both coordinator and C-proc's.

3 Dynamic Partition Algorithm

The web can be partitioned in several ways; in particular, the partition can be obtained from URL-based hash, site-based hash or hierarchically by domain name. The key feature of our model is to conduct dynamic task assignment. This is currently done by means of partition algorithm based on two parameters: **size** of a site and **capacity** of a C-proc. Size of a site may be measured by the number of objects the site holds or the time elapse of last full download of all objects in this site. The size is estimated after the first download, and will be re-estimated after each download. Capacity of a C-proc is proportional to the available resources of an agent. The crawling related resources include CPU, memory, and storage and network bandwidth. According to our design, the partition of URL queues is implemented according to algorithm 3.1.

Algorithm 3.1 Dynamic partition algorithm

```
Input: $S$ is an ordered set of sites according to their sizes
Input: $P$ is an ordered set of C-proc's according to their capacities
Required: $M$ is the mapping list of a pair set (site, C-proc)
Required: $A$ is the list of all agents
Required: $U$ is a list of global URL-queue group by sites, managed by coordinator
Required: URL-queue for each C-proc, maintained by each C-proc
1: Initial assignment: $p_0 \leftarrow s_0$ {assign $s_0$ to $p_0$},$p_1 \leftarrow s_1$,….,
2: add pair $(s_0,p_0),(s_1,p_1)$… to the mapping list $M$;
3: delete the assigned sites from $S$
4: while $|S| > 0$ do
```

```
5:  for all $p_i \in P$, do     /* Check for the tasks of C-proc's
6:      if the task of this C-proc $p_i$ has been finished, do
7:          delete pair $(s_k, p_i)$ from the mapping list $M$
8:          assign $p_i \leftarrow s_0$, delete $s_0$ from queue $S$, add pair $(s_0,p_i)$ to list $M$
9:      end if
10:     if the C-proc $p_i$ found some new URLs not local to its given site $s_j$, do
11:         send the URLs to URL-queue of coordinator $U$
12:     end if
13: end for
14: for all $a_i \in A$, do
15:     if the agent $a_i$ has failed, do
16:         for all C-proc $p_i$ running on this agent, do
17:             add $s_k$ to the head of list $S$ , where $(s_k, p_i)$ is a pair value in $M$
18:             delete pair $(s_k, p_i)$ from the mapping list $M$
19:         end for
20:     end if
21: end for
22: if some new agent is available, do
23:     start several C-proc on this agent
24:     add these C-proc to list $P$, re-order $P$ according to their capacities
25: end if
26: Order the list of global URL-queue $U$ according to their sites
27: for all new URLs $u_i \in U$, suppose the site of $u_i$ is $s_k$, do
28:     if $(s_k, p_i)$ exists in the lists of $M$, do
29:         add new URLs $u_i$ to the URL-queue of C-proc $p_i$
30:     else if $s_k$ is a new discovered site, thus not belonging to $S$, do
31:         add $s_k$ to the tail of list $S$
32:     end if
33: end for
34: end while
```

It should be mentioned that (1) The assignment of $s_k \rightarrow p_i$ means that all URLs of the site s_k are put into URL-queue of C-proc p_i; (2) Step 10-12 indicates that the hyperlinks that are not local to the given site of C-proc are sent to coordinator, then dispatched to the right C-proc by coordinator (step 28-29) or saved in the URL-queue of coordinator for new assignment; (3) The agents is allowed to be dynamically changed in the running. New agents may participate in tasks, while crash agents may be left out.

It is obvious that this kind of assignment may achieve the optimized results because it fully utilizes the available resources. We will discuss this in detail by experiment in section 5.

4 Implementation Issues

To achieve our design goals, we propose a four layers' implementation architecture as shown in figure 2:

- Storage layer: to perform the distributed and parallel file management;
- Scheduling layer: to act as the status monitor and task scheduler;
- Communication layer: to coordinate the data transfer from primary storage to secondary storage and the communication between agents and coordinator;
- Application layer: to perform the tasks of data gathering.

We develop our system based on Globus Toolkit (http://www.globus.org/) and Sun Grid Engine (http://wwws.sun.com/software/gridware/sge.html). Agents and coordinator exchange URLs and control information by MPI (Message Passing Interface). The GridFTP of Globus is used to transfer the large amount of data, such as the data transfer from primary storage to secondary storage. Next we discuss two main issues.

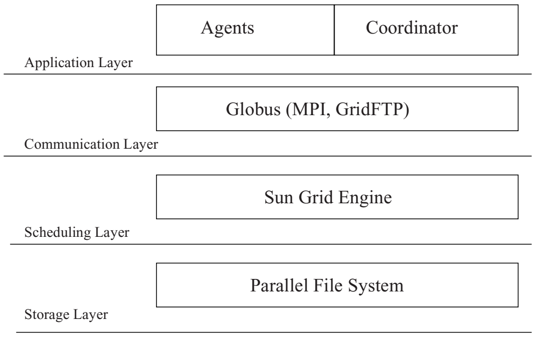

Fig. 2. The Implementation Architecture

Task Scheduling. Sun Grid Engine (SGE) is used to collect the status of grid nodes, and schedule the tasks running on the nodes. Each "Execution Host" acts as an agent, while "Master Host" acts as a coordinator. When data gathering is started, the ordered tasks according to prior knowledge are submitted to "Submit Host". Then the Master Host begins to schedule the tasks according to the resources of all Execution Hosts and SGE's policies that may be configured by the system administrator. When the tasks finish, the Execution Host may release the resources and Master Host will automatically assign new tasks to it. In the running, an agent may join or leave freely. Administration Host manages all register information of nodes, thus its role is the "Monitor". Master Host may re-schedule the tasks according to the register information and status of task execution. The system may also query SGE for the status of all grid nodes, and save them in "Status Info" for further use. Thus, the employee of SGE simplifies the implementation of the system.

Parallel File Management. Web crawler is a heavy I/O system, thus the file management is a key factor that greatly affects its performance. To support dynamic partition technique, we use a virtual central database to hold the data. This central database can be disk array, SAN (Storage Area Network), or Data Grid [8].

At first the data is downloaded into the primary storage. After some time (depend on the amount of data and available storage resources), the downloaded objects may be compressed and transferred back to the secondary storage, namely, central database. This kind of batch communication greatly improves the performance of I/O.

To avoid the overhead of the repeated downloading and analysis of documents that have not been modified, the Data Fetcher uses cached object metadata stored in "Meta-Info" to decide whether to download the documents that are already in the cache. The content of this metadata is represented as <URL, Size, Last-modified>. This metadata has been built into a hash table to facilitate the fast look-ups. According to the partition algorithm 3.1, the assignment of task to C-proc is site based. When one C-proc starts, the coordinator constructs a hash table of cache objects' metadata in the site assigned to the C-proc, and sends this hash table to the C-proc.

The "Data Fetcher" retrieves a URL from the URL-queue, make the HTTP connection, retrieve the URL and analyze the HTTP-header of the response message, and then access the hash table of cached object metadata and check if the URL retrieved from the URL-queue is corresponding to the cached object metadata. If the object is not in the cache or has been modified since last fetching, download the body of the object and store it in primary storage, and send the object to "URL Extractor" and "Convertor" for further processing.

Each C-proc only maintains the metadata of those cached objects, which are local to its current assigned site. Instead of checking for the files on the disk, the Data Fetcher checks the hash table of cached object metadata for the update information of objects. The look-ups of hash table is very fast, thus greatly improves the performance.

Based on the parallel file management, the construction of the parallel crawler is very scalable. All kinds of agents even the diskless workstations are allowed to participate in the crawling task. To ensure batch communication, the C-proc on a diskless agent may hold data in its memory other than local disk.

5 Evaluation

Evaluation model of parallel crawling schemes is described in details in [7]. Here we discuss the parallel properties based on these four key metrics of parallel crawlers.

Overlap. Overlap of downloaded pages is defined as $(N - I) / I$. Here, N represents the total number of pages downloaded by the overall crawler, and I represents the number of unique pages downloaded by the overall crawler. Even in the presence of faults, our model achieves overlap 0, which is optimal, because all C-proc's download the data according to the hash tables of cached objects, and the hash tables are constructed from the updated central database.

Coverage. Coverage of all pages that ought to be downloaded is defined as I / U, where U represents the total number of pages that the overall crawler has to download, and I is the same as in the definition of Overlap. Theoretically, our model achieves coverage 1, which is optimal, even when fault occurs.

If the objects stored locally on a crashed agent have not been transferred back into central database, they will be fetched by the new C-proc on other agents responsible for them. If the objects stored locally on a crashed agent have been transferred back into central database, they will not be fetched by any new C-proc. Because when coordinator starts to assign new task to a C-proc, it will send to the C-proc a hash table of cached objects that belong to the site that the C-proc will be responsible for, as stated in section 4.

Quality. Quality of downloaded pages is defined as $/ A_N \cap P_N \; / \, / \, / P_N \; /$, where A_N stands for the set of the most important N pages that an actual crawler would download, and P_N represents the set of the most important N pages that an ideal crawler would download. Though our crawler uses a parallel per-site breadth-first visit, without dealing with ranking and quality of page issues, it is proved that a breadth-first single-process visit tends to visit high-quality pages first [2,9]. Thus this kind of crawler tends to have a very good performance in quality.

Communication overhead. Communication overhead is defined as *L/N*, where *L* represents the total number of inter-partition URLs exchanged by the overall crawler, and *N* is the same as in the definition of Overlap, represents the total number of pages downloaded by the overall crawler. As it is stated in [2] that on the average every page contains just one link to another site, we have that *n* crawled pages will give rise to *n* URLs that must be potentially communicated to coordinator and other agents. By the definition, the communication overhead is thus no more than 1, which means that the crawler consumes very small network bandwidth for URL exchanges.

To evaluate the performance, we conducted an experiment to demonstrate the advantage of dynamic partition methods. In this experiment, six C-proc's were managed to run on three grid nodes to do the data gathering tasks. They downloaded totally 98585 pages from 30 different sites. We tested four partition methods: "Static-Random" is to partition tasks by static partition, each C-proc was responsible for nearly same number of sites; "Static-Size" is to partition sites according to their objects numbers by static partition, each C-proc tries to be responsible for nearly the same number of objects; "Dynamic-Random" is to partition tasks by dynamic partition algorithm (Algorithm 3.1), the difference is that set of sites *S* is a random set; "Dynamic-Size" is also to partition sites by dynamic partition algorithm, and *S* is an ordered set of sites by their sizes. All C-proc's cooperate together to finish exactly the same tasks each time for different partitions. The workload contributed by one C-proc, called "normalized workload", is measured by the download time of this C-proc divided by the sum of download time of all C-proc's. We use normalized workload rather than the crawling time, because the former is not affected by actual network conditions and server performance of every site while the latter does. Thus it is a good measure for the load balance.

Table 1. The normalization workload of C-proc's under different partition methods

Partition C-Proc	1	2	3	4	5	6	stdev
Static-Randam	**0.27**	**0.20**	**0.09**	**0.14**	**0.16**	**0.14**	**6.19%**
Static-Size	**0.15**	**0.23**	**0.14**	**0.16**	**0.14**	**0.18**	**3.44%**
Dynamic-Random	**0.17**	**0.14**	**0.20**	**0.19**	**0.16**	**0.14**	**2.50%**
Dynamic-Size	**0.17**	**0.16**	**0.17**	**0.18**	**0.16**	**0.16**	**0.82%**

The experimental result is shown in Table 1. The standard variation (stdev) means that the difference of normalized workload distributed in C-proc's. It is observed that the dynamic partition which achieve the smaller standard variation value, is very effective to shorten the difference of download time among C-proc's, thus provides the best of load balance. And the results also demonstrated that the system would achieve better load balance if there exists more prior knowledge.

6 Conclusion

In our work, we address the challenge of designing and implementing a parallel crawler in the context of Grid middleware. We introduce a parallel crawling schema designed using dynamic partition mechanisms to achieve high performance, faulttolerance and scalability, and evaluate our model with the criterions of parallel crawler and performance of load balance. Further work will be included in the near future, such as the employment of page ranking technique to improve the crawling quality of the crawler.

References

1. S. Brin and L. Page: The anatomy of a large-scale hypertextual web search engine. Computer Networks (1998) 107-117
2. P. Boldi, B. Codenotti, M. Santini, and S. Vigna: Ubicrawler: A scalable fully distributed web crawler. In: Proc. AusWeb02. The Eighth Australian World Wide Web Conference (2002)
3. D. Zeinalipour-Yazti, M. Dikaiakos: Design and Implementation of a Distributed Crawler and Filtering Processor. In: A. Halevy, A. Gal (Eds.): Proceedings of the Fifth International Workshop on Next Generation Information Technologies and Systems (NGITS'2002). Lecture Notes in Computer Science, vol. 2382. Springer (2002) 58-74
4. V. Shkapenyuk and T. Suel: Design and implementation of a high-performance distributed Web crawler. In: Proceedings of the 18th International Conference on Data Engineering (ICDE'02). San Jose, CA (2002) 357-368
5. R. L. Walker: Dynamic load balancing model: Preliminary results for parallel pseudo-search engine indexers/crawler mechanisms using MPI and genetic programming. VECPAR 2000. Porto, Portugal (2000) 61-74
6. P. Boldi, B. Codenotti, M. Santini, and S. Vigna: Trovatore: Towards a highly scalable distributed web crawler. In: Proc. of 10th International World Wide Web Conference. Hong Kong, China (2001)
7. J. Cho and H. Garcia-Molina: Parallel crawlers. In: Proc. of the 11th International World–Wide Web Conference (2002)
8. P. Andrews, T. Sherwin and B. Banister: A centralized data access model for grid computing. In: Proceeding of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03). San Diego, California (2003)
9. Marc Najork and Janet L. Wiener: Breadth-first search crawling yields high quality pages. In: Proc. of 10th International World Wide Web Conference. Hong Kong, China (2001)