ParaCA:A Speculative Parallel Crawling Approach on Apache Spark

Yuxiang Li
College of Information Engineering
Henan University of Science and Technology
Luoyang, China
Email: liyuxiang@haust.edu.cn

Zhiyong Zhang
College of Information Engineering
Henan University of Science and Technology
Luoyang, China
Email: xidianzzy@126.com

Bin Liu
College of Information Engineering
Northwest A&F University
Yangling, China
Email: liubin0929@nwsuaf.edu.cn

***Abstract*—The World Wide Web today is growing at a phenomenal rate. The crawling approach is of vital importance to improve the efficiency of crawling the web. The existing sequent crawling algorithms are mostly time consuming and do not support large data well. In order to improve parallelism and efficiency of crawler on distributed network environments, based on the software thread-level speculation technique, this paper proposes a Speculative parallel crawler approach (ParaCA) on Apache Spark. By analyzing the process of web crawler, the ParaCA firstly hires a function to divide a crawling process into several subprocesses which can be implemented independently and then spawns a number of threads to speculatively crawl in parallel. At last, the speculative results are merged to form the final outcome. Comparing with the conventional parallel approach on multicore platform, ParaCA is very efficiency and obtains a high parallelism degree by making the best of the resources of the cluster. Experiments show that the proposed approach could leverage a significant speedup improvement with compare to the traditional approach in average. In addition, with the growing number of working nodes, the execution time decreases gradually, and the speedup scales linearly. The results indicate that the crawling efficiency can be significantly enhanced by adopting this speculative parallel algorithm.**

***Keywords*-crawling approach; parallel; Apache Spark; thread-level speculation; multicore**

I. INTRODUCTION

Crawlers [1] are generally used in search engines to find information that is of interest to users quickly and efficiently from vast Internet information. Crawlers are also used to collect the research data that researchers need. Literature [2] for data acquisition needs, put forward the strategy of fusion of different acquisition programs, the fusion strategy can quickly and efficiently collect large amounts of data. However, with the advent of big data and the advent of Web 2.0, all kinds of information on multimedia social networks have exploded. The efficiency and update speed of single crawlers have been unable to meet the needs of users. Using parallel technology for crawlers can effectively improve the efficiency of crawlers [3], in a big data environment, parallel crawlers are implemented in a distributed architecture, because distributed crawlers are more suitable for large data environments than stand-alone multicore parallel crawlers. Literature [4] used distributed web crawler framework and techniques to collect data from social networking site Sina Weibo to monitor public opinion and other valuable findings, overwhelming traditional web crawlers in terms of efficiency, scalability, and cost, greatly improving the efficiency and accuracy of data collection. Literature [5] put forward a web crawler model of fetching data speedily based on Hadoop distributed system in view of a large of data, a lack of filtering and sorting. The crawler model will transplant single-threaded or multi-threaded web crawler into a distributed system by way of diversifying and personalizing operations of fetching data and data storage, so that it can improve the scalability and reliability of the crawler.

A good approach to process large-scale data is to make use of enormous computing power offered by modern distributed computing platforms (or called big data platforms) like Apache Hadoop [6] or Apache Spark [7, 8]. These popular platforms adopt MapReduce, which is a specialization of the split-apply-combine strategy for data analysis, as their programming model. A standard MapReduce program is composed by pairs of Map operation (whose job is to sorting or filtering data) and Reduce operation (whose job is to do summary of the result by Map operations), and a high parallelism of MapReduce model is obtained by marshalling the operation pairs and performing them in distributed servers in parallel. The platforms that adopt MapReduce model often split the input, turn the large scale problems into sets of problems with small-scale, and then solve the problem sets in a parallel way. At present, many resource-intensive algorithms are successfully implemented to these big data platforms and achieve a better performance [9–11], and it is a good way to enhance conventional algorithms efficiency. However, there exists a problem that the inherent dependence in the conventional crawling approach affects the effect of parallelism. So, designing and implementing a parallel crawling approach with high efficiency on Big Data platforms becomes very essential. By this method, the crawling web data can be split into some data blocks and then captured in parallel. After crawling the webs, the results are validated at a certain point, the proper results would be submitted and the improper ones would be recalculated. Even though false parallelization may occur, it is still a good method to leverage the the crawling performance.

The remaining parts of this paper are organized as follows:we first briefly describe the execution model and related work in section II; we present the framework of ParaCA in section III; based on the framework, we present the implementation of ParaCA in section IV; section V presents the evaluation of ParaCA; Then experimental process and results are shown in section VI. Finally, conclusion is present in section VII.

II. Execution Model and Related Work

This section mainly presents the execution model and related works. The study object in the execution model is crawling approach [12], which is the most popular and has been intensively applied. In addition, as the measure to handle the dependence in crawling approach, the speculative multithreading is also introduced. Finally, Apache Spark is chosen as the computing platform to implement the speculative parallel algorithm for its powerful computation ability.

A. Apache Spark Parallel Computing Platform

In order to decompress large-scale data in parallel with STLS technique, the Apache Spark [13] is chosen as the computing platform. Apache Spark is a distributed computing platform developed at the Berkeley AMPLab [14]. Different from other computing clusters with multiprocess model like Apache Hadoop, Apache Spark adopts the multithreading model, which makes Apache Spark provide a good support on STLS technique. Apache Spark is a classic master/workers mode in which the master distributes tasks to workers, while workers are responsible for executing tasks. Fig. 1 is a demonstration on Spark multithreading model.

B. Related Works

In order to obtain the micro-blogging data quickly, Literature [15] extended the parallel framework based on the single-process crawler and realized parallel data capture function based on MPI. The parallel crawler has a good speedup ratio and can quickly obtain data, and these data with real-time and accuracy. Literature [16] proposed a Kademlia-based fully distributed crawler clustering method. According to the XOR characteristics of Kademlia and the available resources of nodes, a complete distribution with task allocation, exception handling, node join and exit processing, and a crawler cluster model with load balancing was built. Literature [16] studied a distributed Hadoop-based crawler technology and implemented parallel processing of reptiles on the basis of the distributed crawler. The slave nodes not only process all the sub-tasks assigned by the master node in parallel among nodes, but also Multi-threaded internal tasks are also processed internally from within the node.

Regarding the refresh strategy of incremental crawler pages, Zhou et al. [15] used sampling samples to determine the refresh time. Since the update frequency of different sites is not the same, you can use this difference for coarse-grained packet sampling, you can also use the characteristics of web page changes for the fine-grained packet sampling. Schonfeld et al. [17] used the last modification time of the web pages in the site metadata to select the pages to be refreshed. The web server generally stores all the URLs in the server and the last modification time as metadata. Literature [18, 19] proposed a page refresh strategy based on Poisson distribution as a page refresh method for incremental crawler. A large number of studies have proved that webpage changes generally follow the Poisson process, and according to this rule, a refresh model can be established for webpage changes to predict the next update time of the webpage. C Olston et al. [20] proposed a refresh strategy based on information cycle, which combined the sampling based on webpage features with such periodic changes conforming to Poisson distribution, and dynamically adjusted the refreshing cycle of web pages according to the upper and lower utility value boundaries. Literature [15] improved Super-shingle algorithm based on C Olston et al., making it suitable for video resource crawlers. Since C Olston et al. introduced the method of estimating utility values without any practical significance, a practical and effective method of estimating utility thresholds was given in [20]. Compared with the previous border-based methods, this utility-based method better balances freshness and refresh costs, achieving better freshness at a lower cost. Pavai G. et al. [21] proposed an incremental crawler based on probabilistic method to deal with the dynamic changes of surface web pages. The method of predicting the probability of web page variation based on Bayesian theory was modified to deal with deep web dynamic changes. K Gupta et al [22] proposed an accuracy-aware crawling techniques for cloud crawler that allowed local data to be re-crawled in a resource / budget-constrained environment to retrieve the maximum amount of information with high accuracy.

In summary, the existing crawler technology has not been in a good balance of efficiency, refresh cost and freshness, so this paper presents a distributed parallel crawler to improve the efficiency of crawlers, and proposes an incremental update algorithm based on time-aware to get better freshness at a lower refresh cost, and combines the two to better

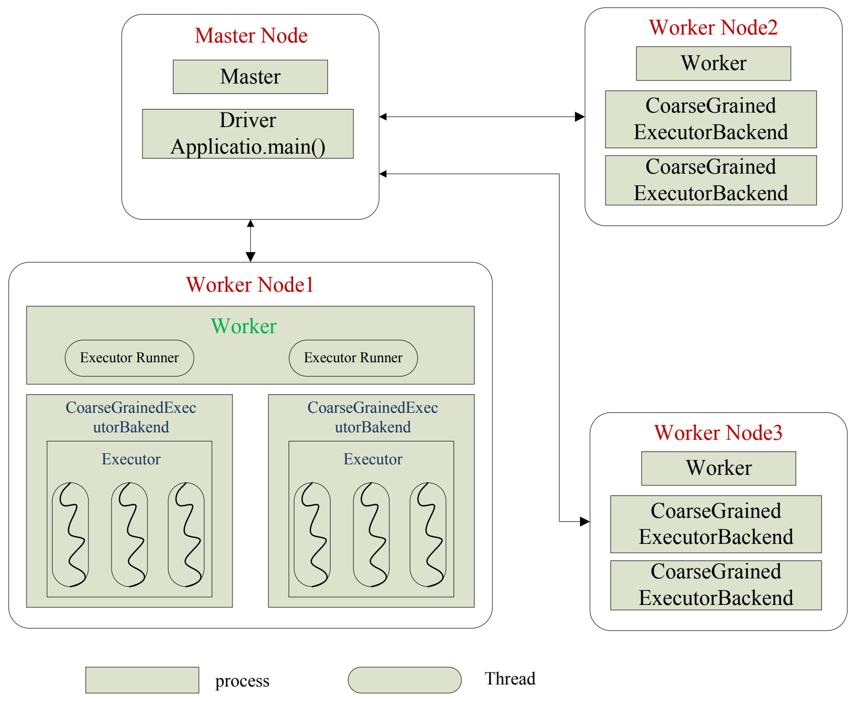

Master Node
Master
Driver
Applicatio.main()
Worker Node2
Worker
CoarseGrained ExecutorBackend
CoarseGrained ExecutorBackend
Worker Node1
Worker
Executor Runner
Executor Runner
CoarseGrainedExecutorBakend
Executor
CoarseGrainedExecutorBakend
Executor
Worker Node3
Worker
CoarseGrained ExecutorBackend
CoarseGrained ExecutorBackend
process
Thread

Figure 1. The Multithreading Model of Apache Spark

balance efficiency, refresh costs and freshness.

III. Spark-based Distributed Parallel Crawler Framework

The distributed crawler system used in this paper adopts a master-slave structure. That is, one master node controls all slave nodes to perform crawl tasks. The master node is responsible for allocating tasks and ensures load balancing of all slave nodes in the cluster. The used allocation algorithm is to calculate the hash value of the host corresponding to each URL, and then divide the URL of the same host into a partition. The purpose of this is to have the URL of the same host crawled on one machine. Distributed crawlers can be viewed as a combination of multiple centralized crawler systems. Each slave node is equivalent to a centralized crawler system. These centralized crawler systems are controlled and managed by a master node in a distributed crawler system.

From the diagram of distributed parallel crawler framework in Fig.2, we can see that the main part of the framework includes master node for task generation, task allocation and scheduling, and the master node′s control and management of the entire system (such as the depth of crawler, configuration of update time, system startup and stop etc.) ; The crawler cluster is responsible for parallel downloading pages; the Map/Reduce function module is responsible for parsing pages, optimizing links, and web page updates; Message middleware is responsible for communication and collaboration between master node, crawler nodes, and clusters (e.g. log management, data exchange and maintenance between clusters, etc.); and Distributed File System (HDFS) for data storage.

A. *Parallel Crawling*

To process large-scale data stored in HDFS in parallel, Hadoop offers a parallel computing framework called Map / Reduce. Spark is founded on Hadoop. The framework effectively manages and schedules nodes in the entire cluster to complete the programs′ parallel execution and data processing and allows every slave node to localize calculation data on the local node as much as possible.

As can be seen from Fig.2, the core of the entire crawler system can be divided into three modules, including download module, parsing module and optimization module. Every module is an independent function module, and every module corresponds to a Map / Reduce process.

- The download module can download web pages in parallel. Specific download is completed in the Reduce phase, and multi-threaded download is used.
- Parsing module can analyze downloaded pages in parallel, extract the link out. The module not only needs a Map stage to complete the goal, but also limits the type of links to prevent the extracted links to other sites through the rules.
- The optimization module can optimize the collection of links in parallel and filter out duplicate links.

It can be seen that the parallelism of the Web crawler system is achieved through these three parallelizable mod-

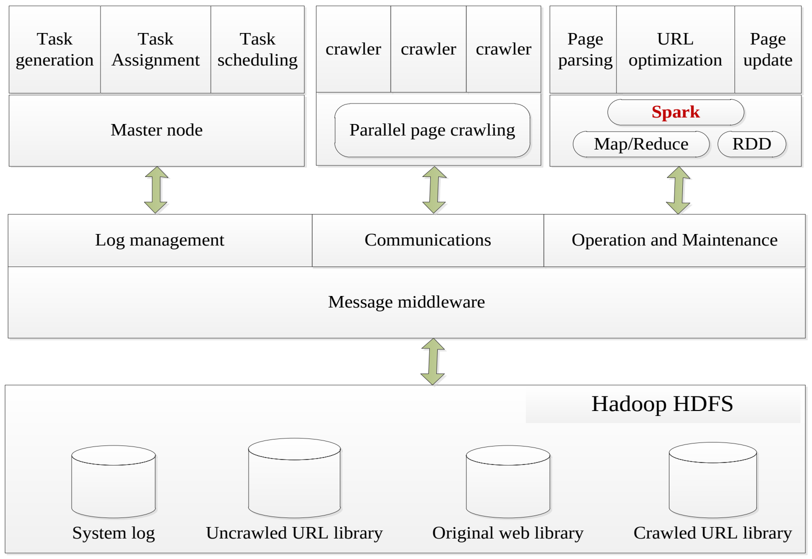


Figure 2. Distributed Parallel Crawler Framework Diagram

ules, which are essentially implemented through the parallel computing framework of Map / Reduce.

At the beginning of the Map / Reduce task, the input data is split into several slices, with a default of 64 MB for every slice. Every piece is processed by a Map process, a crawler can open multiple Map processes at the same time. After the output of all Map is combined, according to the partitioning algorithm, the URL of the same site is assigned to a partition, so that the URL of the same site can be crawled on the same machine, the tasks of every partition are processed by a Reduce process, several partitions have several Reduces which perform parallel processing, while a crawler can also open multiple Reduce processes. Finally, the results of the parallel execution are saved to HDFS.

Fig.3 shows a parallel crawler framework diagram with Spark, in which three critical modules are *URL_Download()*, *URL_Parser()*, *URL_Optimization()*. "*Seeds_File.txt*" is the source of URLs, which are used to the process of initial crawler. From the file of "*Seeds_File.txt*", one new URL is extracted and then used to judge whether its length is larger than 0 or not. If the length of extracted URL is greater than 0, the next step is to download its web page. The downloaded pages need to be parsed, including *URL_Parser* and *Content_Parser*. The parsed contents need to be optimized, so to filter the improper URLs."sc = *SparkContext*(*appName*="URL*Download*")" and "*urls* = *sc.parallelize*(*new_urls*)" are used to realize the parallelization of download process. After this process, "*url_html_list*" is generated and used for parsing URLs and contents.

Fig.4 shows a parallel download diagram with Spark, in which three critical processes are included, *i.e.* judgement of urls, using SparkContext to get *sc*, using *parallelize()* to realize parallelization.

Fig.5 presents a parallel parser diagram with Spark, in which three critical processes are also included, *i.e.* URL_download, using SparkContext to get *sc*, using *parallelize()* to realize parallelization of *url_html_list*.

Similar to Fig.4 and Fig.5, Fig.6 shows the process of parallelizing URL_optimization, including URL_parser, obtaining *sc*, using *parallelize(url_urls_list)* to realize the parallelization. The function *collect()* is used to obtain new url list, and the function *distinct_urls* is used to remove the repetitive urls.

The specific processes of parallelizing crawling can be reduced to be:

(1) Collecting a set of seeds. First, for each crawler target to collect a URL seed as the entrance link to download data, and then the files of seeds from the local file system upload to input folder of hadoop cluster distributed file system, input folder always holds the URL to be crawled by the current layer. At the same time, the setting layer which has been crawled is 0.

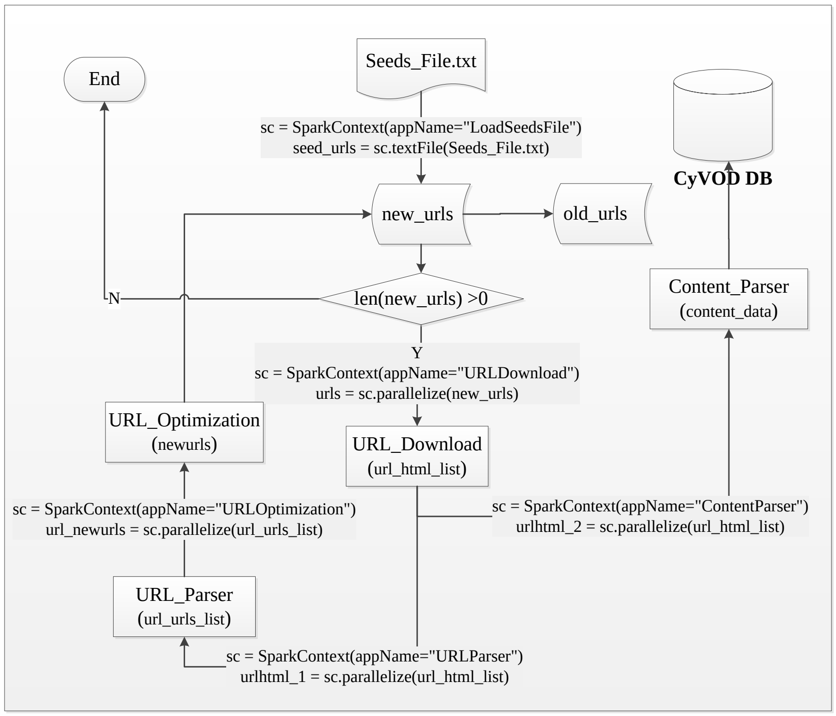
(2) Judge whether the list to be fetched in the input folder is empty. If yes, skipping to (7); otherwise, executing (3).

(3) Download pages in parallel. And save the original page to the html folder in HDFS, html folder holds raw web pages of every layer.

(4) Parse pages in parallel. Extract the eligible links from the crawled pages in the html folder and save the results to the output folder in HDFS. The output folder always stores the outgoing links that are parsed at the current level.

(5) Optimize outgoing links in parallel. Filter out the crawled URLs from all the parsed URLs in the output folder, and save the optimized results to input folder in HDFS for the next crawl.

(6) Judge whether the number of crawled layers is less than


Seeds_File.txt
End
CyVOD DB
new_urls
old_urls
len(new_urls) >0
N
Y
Content_Parser (content_data)
URL_Optimization (newurls)
URL_Download (url_html_list)
URL_Parser (url_urls_list)

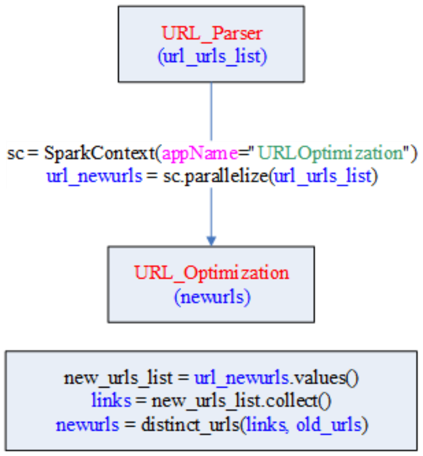
```
sc = SparkContext(appName="LoadSeedsFile")
seed_urls = sc.textFile(Seeds_File.txt)

sc = SparkContext(appName="URLDownload")
urls = sc.parallelize(new_urls)

sc = SparkContext(appName="URLOptimization")
url_newurls = sc.parallelize(url_urls_list)

sc = SparkContext(appName="ContentParser")
urlhtml_2 = sc.parallelize(url_html_list)

sc = SparkContext(appName="URLParser")
urlhtml_1 = sc.parallelize(url_html_list)
```

Figure 3. Parallel Crawler Framework Diagram with Spark

the parameter depth. If yes, "crawled layers" increase by 1, return (2); otherwise enter (7).
(7) Combine the pages crawled by every layer and remove the duplicate crawled pages. The results are still stored in the html folder.
(8) According to the webpage crawling plan, these pages with the crawling task at the moment are added to the crawling list.
(9) Further parse webpages content. Analyze the content of the webpages in parallel, and then parse out the required attribute information from the merged and duplicated webpages. The attribute information required by the system includes title, publishing time, copyright owner, text, and video source.
(10)According to the attribute information which is parsed out, further screening is done. If the attribute information satisfies the user rules, such as the publication date in the last 7 days and the content of the text related to the scientific and technical information. It will upload the attribute information that meets the conditions, including the URLs, to the server database. Otherwise, give up.

B. Parallel algorithm based Map/Reduce

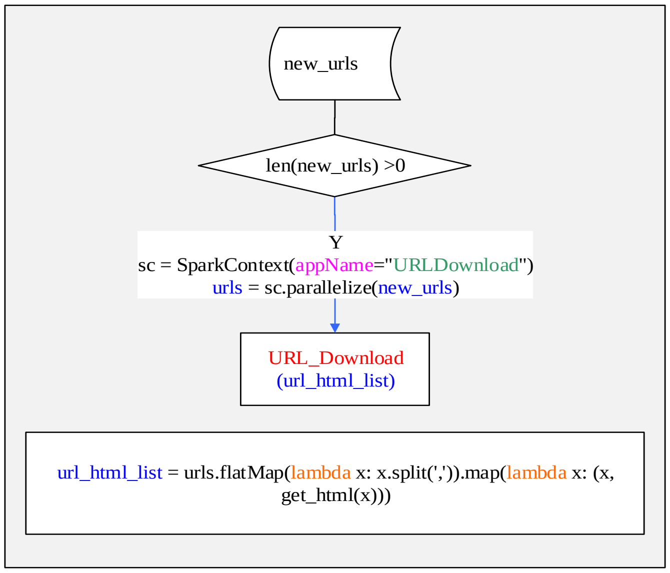
Definition 1: Crawler = $\{c_1, c_2, ..., c_n\}$: represents a collection of crawler nodes in a cluster. c_i represents the i_{th} crawler node. The maximum number of Map processes and the maximum number of Reduce processes which a reptile



```
sc= SparkContext(appName="URLOptimization")
url_newurls = sc.parallelize(url_urls_list)

new_urls_list = url_newurls.values()
links = new_urls_list.collect()
newurls = distinct_urls(links, old_urls)
```

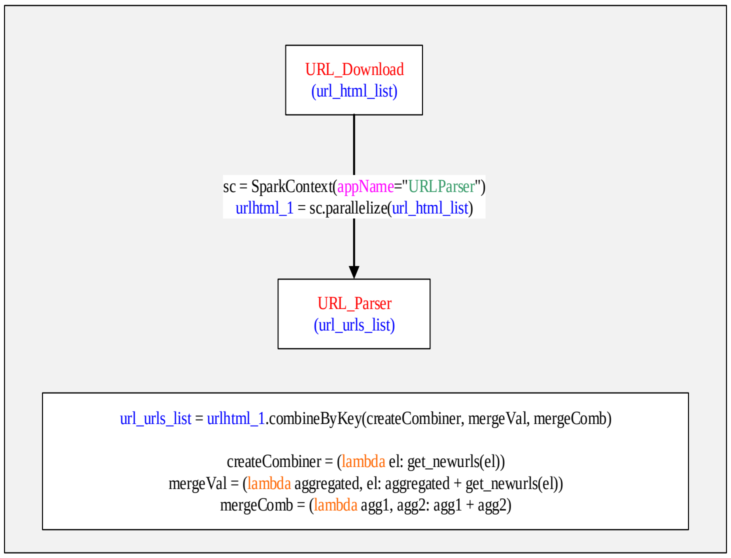
Figure 6. Parallel Optimization with Spark

node can open are determined by the number of processors on the node.
Definition 2:$\{split_0, split_1 ,..., split_{m-1}\}$: represents a collection of file slices. A slice is handled by a Map process.
Definition 3:$\{part_0, part_1 ,..., part_{k-1}\}$: represents a collection of file partitions. A partition is handled by a Reduce process.
Assuming $m = 2n$, $k = n$, then parallel algorithm based



```
sc = SparkContext(appName="URLDownload")
urls = sc.parallelize(new_urls)
url_html_list = urls.flatMap(lambda x: x.split(',')).map(lambda x: (x, get_html(x)))
```

Figure 4. Parallel Download with Spark



```
sc = SparkContext(appName="URLParser")
urlhtml_1 = sc.parallelize(url_html_list)
url_urls_list = urlhtml_1.combineByKey(createCombiner, mergeVal, mergeComb)

createCombiner = (lambda el: get_newurls(el))
mergeVal = (lambda aggregated, el: aggregated + get_newurls(el))
mergeComb = (lambda agg1, agg2: agg1 + agg2)
```

Figure 5. Parallel Parser with Spark

Map/Reduce is shown in Algorithm 1.

In Table I, a parallel algorithm (Algorithm 1) based Map/Reduce is specifically introduced. Firstly, an input-file is splitted into *m-1* parts; Then, send an adjacent slices (*split_k*, *split_k+1*, where *k*%2==0) to *c_k*, and *split_k* ∼ *split_k+1* are all defined above; Next, map processes are performed to process these adjacent slices, and combine all map output to *Inter-results*; Then, use the partitioner to partition *Inter-results* to $part\_0 \sim part\_k - 1$ (k∈N); Next, send $part\_i$ to $c\_i + 1$(i∈N), and open a process $part\_j \Rightarrow partition\_j$ (j∈N)

IV. Experiment and Analysis

A. Experiment Configuration

This section will present a performance evaluation of ParaCA. Table VI shows the specific configuration of experiment environment.

B. Analysis of Experimental Results

Fig.6 shows a time comparison between sequential crawling and parallel crawling. The left green line represents the time of sequential crawling while the right green line represents the time of parallel crawling.

Table I
ALGORITHM 1:PARALLEL ALGORITHM BASED MAP/REDUCE

Algorithm 1:Parallel algorithm based Map/Reduce

```
Input: Input-File
Output: Output-File
1: Begin
2: Split Input-File into m slices =>{split0, split1, ..., splitm-1};
3: send split0, split1 to c1 , open two Map processes to process this two slices;
4: and send split2, split3 to c2 , open two Map processes to process this two slices;
5: and ....;
6: and send splitm-2, splitm-1 to cn , open two Map processes to process this two slices;
7: Combine all Map output => Inter-results;
8: partitioner(Inter-results) => {part0, part1, ..., partk-1};
9: send part0 to c1 , open a Reduce process to process part0 => partition_0;
10: and send part1 to c2, open a Reduce process to process part1 => partition_1;
11: and ....;
12: and send partk-1 to cn , open a Reduce process to process partk-1 => partition_n-1 ;
13: Output-File = {partition_0,partition_1,...,partition_n-1};
14: End
```

Table II
ALGORITHM 1:PARALLEL DOWNLOAD OF URL BASED SPARK

Algorithm 2:Parallel Download of URL based Spark

```
def urls_download(urls):
sc = SparkContext(appName="URLDownload");
new_urls = sc.parallelize(urls);
url_html_list = new_urls.flatMap(lambda x: x.split(',')).map(lambda x: (x, get_html(x)));
output = url_html_list.collect();
sc.stop();
print('urls_download: %s ' % len(output));
return output;
```

Table III
ALGORITHM 1:PARALLEL PARSER OF URL BASED SPARK

Algorithm 3:Parallel Parser of URL based Spark

```
def url_parser(url_html_list):
sc = SparkContext(appName="URLParser");
url_htmls = sc.parallelize(url_html_list);
createCombiner = (lambda el: get_newurls(el));
mergeVal = (lambda aggregated, el: aggregated + get_newurls(el));
mergeComb = (lambda agg1, agg2: agg1 + agg2);
new_urlss = url_htmls.combineByKey(createCombiner, mergeVal, mergeComb);
output = new_urlss.collect(); sc.stop();
print('url_parser: %s %s' len(output), output);
return output;
if_name_ == "_main_":
sc = SparkContext(appName="LoadSeedsFile");
seeds_file = "file:///home/hadoop/CyCrawler/News_CyCrawler/Spider/seeds-file.txt";
lines = sc.textFile(seeds_file);
root_urls = lines.flatMap(lambda x: x.split(' ')).distinct();
urls = root_urls.collect();
sc.stop();
url_html_list = urls_download(urls);
url_parser(url_html_list);
```

Table IV
ALGORITHM 4:PARALLEL CONTENT PARSER BASED SPARK

Algorithm 4:Parallel Content Parser based Spark

```
def content_parser(url_html_list):
sc = SparkContext(appName=”ContentParser”);
url_htmls = sc.parallelize(url_html_list);
createCombiner = (lambda el: get_content(el));
mergeVal = (lambda aggregated, el: aggregated + get_content(el));
mergeComb = (lambda agg1, agg2: agg1 + agg2);
new_data = url_htmls.combineByKey(createCombiner, mergeVal, mergeComb);
output = new_data.collect();
sc.stop();
return output;
if _name_ == ”_main_”:
old_urls = [];
sc = SparkContext(appName=”LoadSeedsFile”);
seeds_file = ”file:///home/hadoop/CyCrawler/News_CyCrawler/Spider/video-file.txt”;
lines = sc.textFile(seeds_file);
root_urls = lines.flatMap(lambda x: x.split(' ')).distinct();
urls = root_urls.collect();
print(urls);
sc.stop();
for url in urls:
old_urls.append(url);
 print(”======”);
print(urls);
url_html_list = urls_download(urls);
content_parser(url_html_list);
```

Table V
ALGORITHM 5:PARALLEL OPTIMIZATION OF URL BASED SPARK

Algorithm 5:Parallel Optimization of URL based Spark

```
def distinct_urls(links, old_urls):
new_urls = [];
num = 0;
for j in range(len(links)):
if links[j] = None:
num = num + len(links[j]);
for url in links[j]:
if url not in old_urls:
if len(url)>0:
new_urls.append(url);
print(′before url_optimi: %s ′ % num);
return list(set(new_urls));
def url_optimi(url_urls_list, old_urls):
sc = SparkContext(appName=”URLOptimization”);
url_urls_list2 = sc.parallelize(url_urls_list);
new_urls_list = url_urls_list2.values();
links = new_urls_list.collect();
sc.stop();
urls = distinct_urls(links, old_urls);
print(′after url_optimi: %s ' % len(urls), urls);
return urls;
if _name_== ”_main_”:
old_urls = [];
sc = SparkContext(appName=”LoadSeedsFile”);
seeds_file = ”file:///home/hadoop/CyCrawler/News_CyCrawler/Spider/seeds-file.txt”;
lines = sc.textFile(seeds_file);
root_urls = lines.flatMap(lambda x: x.split(' ')).distinct();
urls = root_urls.collect();
print(urls);
sc.stop();
for url in urls:
old_urls.append(url);
url_html_list = urls_download(urls);
url_newurls_list = url_parser(url_html_list);
url_optimi(url_newurls_list, old_urls);
```

Table VI
CONFIGURATION OF EXPERIMENT ENVIRONMENT

Item	**Configuration**
Servers	Lenovo System x3850 x6 (2 sets), Lenovo SR590 (2 sets), IBM System X3500 M4;
Number of Cores	120
Operation System	CentOS, Ubuntu
Parallel Platform	Haddop2.7.0, Spark2.3.0

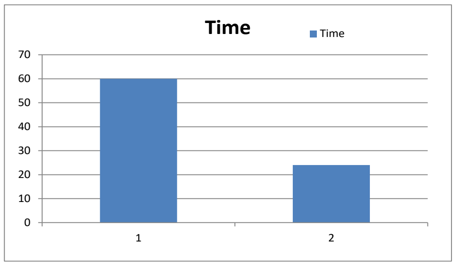


Figure 7. Comparison of Crawling Time

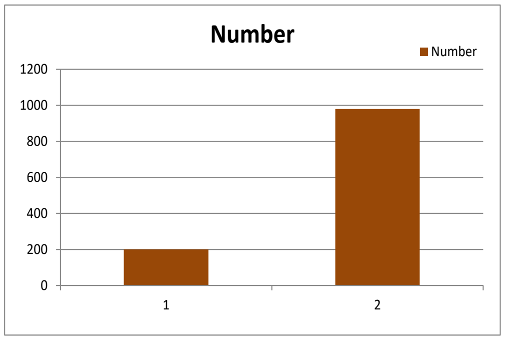


Figure 8. Comparison of Crawling Websites

Fig.8 shows a comparison of crawling websites between sequential crawling and parallel crawling. The left orange line represents the number of sequential crawling websites while the rigth orange line represents the number of parallel crawling websites.

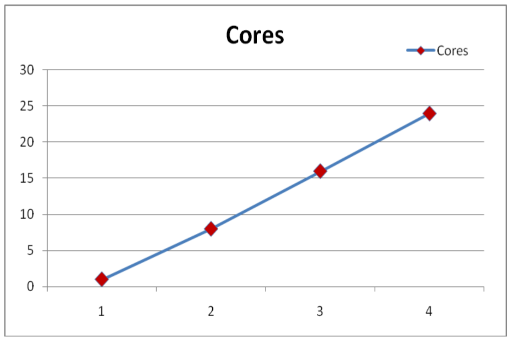


Figure 9. Number of Cores

Fig.9 shows the changing of core number.

ACKNOWLEDGEMENT

We thank all members of Henan International Joint Laboratory of Cyberspace Security Applications for their great support, and give our best hope to them for their collaboration. We also thank reviewers for their careful comments and suggestions. The work was sponsored by National Natural Science Foundation of China Grant No.61972133, Project of Leading Talents in Science and Technology Innovation for Thousands of People Plan in Henan Province Grant No.204200510021, Henan Province Key Scientific and Technological Projects Grant No.192102210130 and No.202102210162, and Key Scientific Research Projects of Henan Province Universities Grant No.19B520008.

REFERENCES

[1] D. M. Zhou, "Survey of high-performance web crawler," *Computer Science*, 2009.
[2] X. U. Yan-Fei, Y. Liu, and W. U. Wen-Peng, "Research and application of social network data acquisition technology," *Computer Science*, 2017.
[3] R. Guo, H. Wang, M. Chen, J. Li, and H. Gao, "Parallelizing the extraction of fresh information from online social networks," *Future Generation Computer Systems*, vol. 59, no. C, pp. 33–46, 2016.
[4] J. Xia, W. Wan, R. Liu, G. Chen, and Q. Feng, "Distributed web crawling: A framework for crawling of micro-blog data," in *International Conference on Smart and Sustainable City and Big Data*, 2016, pp. 62–68.
[5] L. Su and F. Wang, "Web crawler model of fetching data speedily based on hadoop distributed system," in *IEEE International Conference on Software Engineering and Service Science*, 2017, pp. 927–931.
[6] P. S. Honnutagi, "The hadoop distributed file system," *International Journal of Computer Science & Information Technolo*, 2014.
[7] A. G. Shoro and T. R. Soomro, "Big data analysis: Apache spark perspective," vol. 15, 2015.
[8] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, and M. J. Franklin, "Apache spark: a unified engine for big data processing," *Communications of the Acm*, vol. 59, no. 11, pp. 56–65, 2016.
[9] R. Z. Qi, Z. J. Wang, and S. Y. Li, "A parallel genetic algorithm based on spark for pairwise test suite gen-

eration," *Journal of computer science and technology*, vol. 31, no. 2, pp. 417–427, 2016.

[10] H. Qiu, R. Gu, C. Yuan, and Y. Huang, "Yafim: A parallel frequent itemset mining algorithm with spark," in *Parallel & Distributed Processing Symposium Workshops*, 2014, pp. 1664–1671.

[11] K. Shi, J. Denny, and N. M. Amato, "Spark prm: Using rrts within prms to efficiently explore narrow passages," in *IEEE International Conference on Robotics and Automation*, 2014, pp. 4659–4666.

[12] G. M. Siddesh, K. Suresh, K. Y. Madhuri, M. Nijagal, B. R. Rakshitha, and K. G. Srinivasa, "Optimizing crawler4j using mapreduce programming model," *Journal of the Institution of Engineers*, vol. 98, no. 3, pp. 1–8, 2016.

[13] Z. Wang, Y. Zhao, Y. Liu, Z. Chen, C. Lv, and Y. Li, "A speculative parallel decompression algorithm on apache spark," *Journal of Supercomputing*, vol. 73, no. 9, pp. 1–30, 2017.

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Usenix Conference on Hot Topics in Cloud Computing*, 2010, pp. 10–10.

[15] Z. H. Zhou, H. R. Zhang, and J. Xie, "Data crawler for sina weibo based on python," *Computer Application*, vol. 34, no. 11, pp. 3131–3134, 2014.

[16] Z. M. Huang, X. W. Zeng, J. Cheng, and G. University, "Method for fully distributed crawler cluster based on kademlia," *Computer Science*, 2014.

[17] U. Schonfeld and N. Shivakumar, "Sitemaps:above and beyond the crawl of duty," in *International Conference on World Wide Web, WWW 2009, Madrid, Spain, April*, 2009, pp. 991–1000.

[18] T. Meng, J. M. Wang, and H. F. Yan, "Web evolution and incremental crawling," *Journal of Software*, vol. 17, no. 5, pp. 1051–1067, 2006.

[19] M. C. Calzarossa and D. Tessera, "Modeling and predicting temporal patterns of web content changes," *Journal of Network & Computer Applications*, vol. 56, pp. 115–123, 2015.

[20] C. Olston and S. Pandey, "Recrawl scheduling based on information longevity," in *Proceeding of the International Conference on World Wide Web*, 2008, pp. 437–446.

[21] G. Pavai and T. V. Geetha, "Improving the freshness of the search engines by a probabilistic approach based incremental crawler," *Information Systems Frontiers*, pp. 1–16, 2017.

[22] K. Gupta, V. Mittal, B. Bishnoi, S. Maheshwari, and D. Patel, "Act: Accuracy-aware crawling techniques for cloud-crawler," *World Wide Web-internet & Web Information Systems*, vol. 19, no. 1, pp. 69–88, 2016.