SpecCA:A Parallel Crawling Approach based on Thread Level Speculation

Yuxiang Li * and Yaning Su

Posted Date: 30 October 2024

doi: 10.20944/preprints202410.2331.v1

Keywords: crawling approach; parallel; Apache Spark

 doi:10.20944/preprints202410.2331.v1

Article

SpecCA:A Parallel Crawling Approach based on Thread Level Speculation

Yuxiang Li [1,2,3,*] and Yaning Su [1,2,3]

[1] College of Information Engineering, Henan University of Science and Technology, Kaiyuan Avenue, Luoyang, 471023, Henan Province, China, liyuxiang19841203@163.com

[2] Henan International Joint Laboratory of Cyberspace Security Applications, Henan University of Science and Technology, Kaiyuan Avenue, Luoyang, 471023, Henan Province, China, liyuxiang19841203@163.com

[3] Henan Intelligent Manufacturing Big Data Development Innovation Laboratory, Henan University of Science and Technology, Kaiyuan Avenue, Luoyang, 471023, Henan Province, China, liyuxiang19841203@163.com

* Correspondence: liyuxiang19841203@163.com; 15038590339

Abstract: The World Wide Web today is growing at a phenomenal rate. The crawling approach is of vital importance to leverage the efficiency of web crawling. The existing crawling algorithms on multicore platforms easily suffer from time consuming and can not support large data well. In order to exploit the potential parallelism and efficiency of crawling on Spark, based on the software thread level speculation, this paper proposes a Speculative parallel crawler approach (SpecCA) on Apache Spark. By analyzing the process of web crawler, the SpecCA firstly hires a function to divide the whole crawling process into several subprocesses which can be implemented independently and then spawns a number of threads to speculatively implement every subprocess in parallel. At last, the speculative results are merged to form the final outcome. Compared with the conventional parallel approach on multicore platform, SpecCA is very efficiency and leverages a high parallelism degree by adequately using the resources of the cluster. Experiments show that SpecCA could achieve a significant speedup improvement with compare to the traditional approach in average. Additionally, with the growing number of working nodes, the execution time decreases gradually while the speedup scales linearly. The results indicate that the efficiency of web crawling can be significantly enhanced by adopting this speculative parallel algorithm.

Keywords: crawling approach; parallel; Apache Spark

1. Introduction

Crawlers[1, 2] are generally utilized in search engines to search potential 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. For data acquisition, Literature put forward the strategy of fusing different acquisition programs, in which 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 crawler have been unable to meet the needs of users. Using parallel technology for crawlers can effectively improve the crawlers' efficiency, 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 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 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 singlethreaded or multi-threaded web crawler into a distributed system by way of diversifying and

 doi:10.20944/preprints202410.2331.v1

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 or Apache Spark[8, 9]. 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[10-12], 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. With 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 performance of crawling.

The remaining parts of this paper are organized as follows:the execution model and related work are described in section 2; Thread Level Speculation-based Distributed Crawer is presented in section 3; The experiment and analysis is shown in section 4. Finally, conclusion is present in section 5.

2. 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, which is the most popular and has been intensively applied. In addition, as the measure to handle the dependence in crawling approach, the Thread Level Speculation(TLS) is also introduced. Finally, Apache Spark is chosen as the computing platform to implement the speculative parallel algorithm for its powerful computation ability.

2.1. Thread Level Speculation

As an effective mechanism, Thread Level Speculation(TLS)[15, 16] parallelizes irregular programs to realize the improvement of speedups. With an execution model of TLS, sequential programs are performed partition into respective multiple threads, each of which does execution of one part of the sequential programs. Among all of concurrently executed threads, non-speculative thread is a special thread which is the only one thread allowed to commit results to the shared memory. The other threads are all speculative except this thread. A spawning instruction pair marks the speculative thread. On program execution, when a spawning instruction is recognized and if spawning is allowed by the existing processor resources, a new speculative thread is to be spawned by its parent thread. After the non-speculative thread finishes execution, it need verify whether or not the generated data of its successor thread is correct. If the validation proves to be true, all generated values will be committed to memory by the non-speculative thread to memory, then the successor thread turns to be non-speculative. If not, the non-speculative threads would make all speculative successor threads revoked and re-executed.

On Prophet, SP and CQIP constitute the spawning instruction pair. During program execution, a new thread which executes the code segment following the CQIP speculatively is generated from the SP in a parent thread. Figure 1 shows the execution model of TLS. A speculative multithreaded program is generated by mapping sequential programs with SPs and CQIPs. If SPs and CQIPs are ignored, the speculative multithreaded program turns to be one sequential program, which is illustrated in Figure 1(a). When there appears one SP in parent thread on program execution and

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 30 October 2024 doi:10.20944/preprints202410.2331.v1

there exist idle cores, a new speculative thread will be spawned and the code segment following the CQIP will be executed speculatively, and Figure 1(b) shows the process. What leads to speculation failure is Read after Write (RAW) violations or validation failure. When validation failure happens, a parent thread makes the speculative threads executed in a sequential manner, and the process is shown in Figure 1(c). Figure 1(d) shows that there is a RAW dependence violation, in which the speculative thread restarts itself upon its current state.

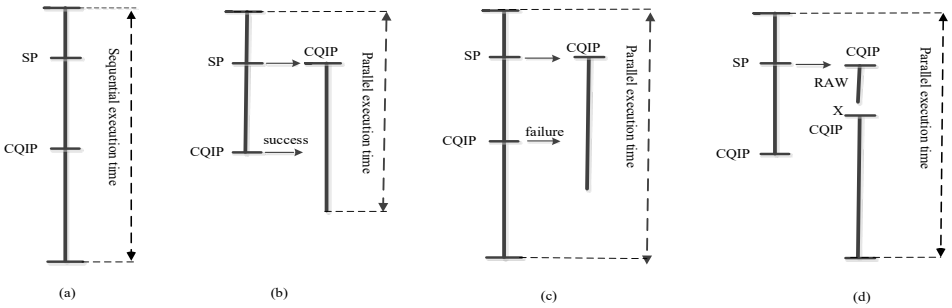
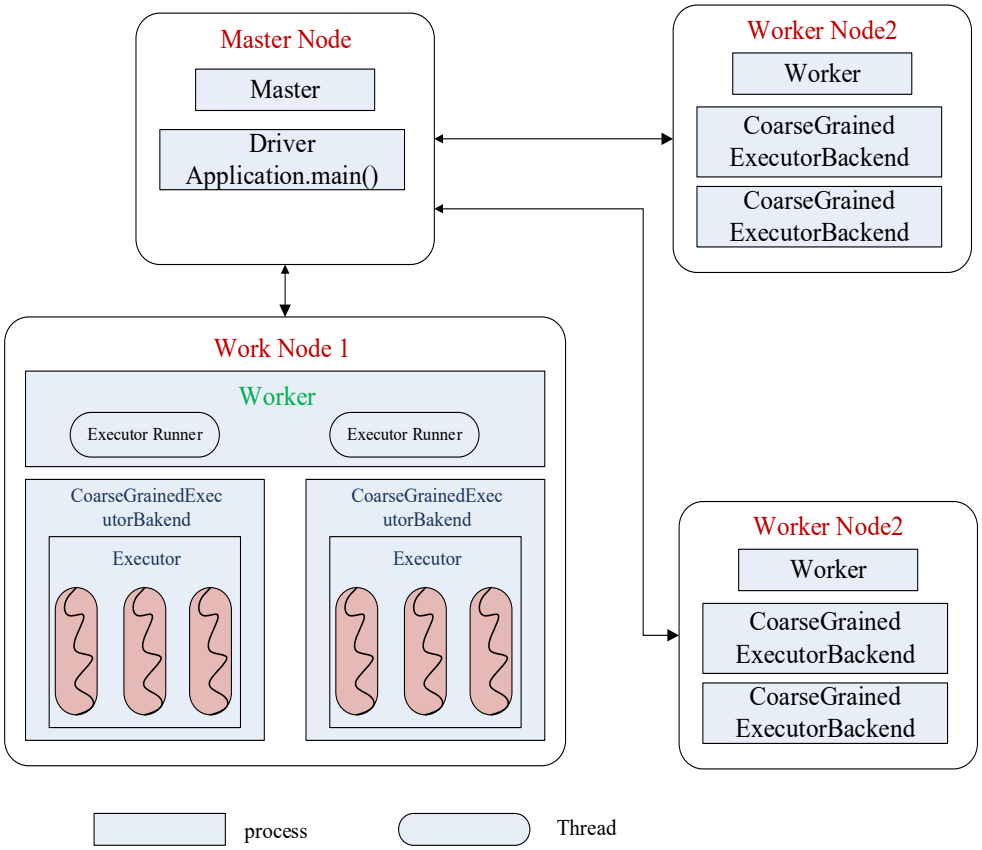


Figure 1. Models for TLS: (a) Sequential Execution; (b) Successful Parallel Execution; (c) Failed Parallel Execution; (d) Read-After-Write Violation.

2.2. Parallel Computing Platform-Apache Spark

In order to decompress large-scale data in parallel with Software Thread Level Speculation(STLS), the Apache Spark is chosen as the computing platform. Apache Spark is a distributed computing platform developed at the Berkeley AMPLab. 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. Figure 2 is a demonstration on Spark multithreading model.



 doi:10.20944/preprints202410.2331.v1

Figure 2. The Multithreading Model of Apache Spark.

2.3. Related Works

In order to obtain the micro-blogging data quickly, Literature 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 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 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 processed all the sub-tasks assigned by the master node in parallel among nodes, but also Multi-threaded internal tasks were also processed internally from within the node.

Regarding the refresh strategy of incremental crawler pages, Zhou et al. used sampling samples to determine the refresh time. Since the update frequency of different sites was not the same, you could use this difference for coarse-grained packet sampling, and could also use the characteristics of web page changes for the fine-grained packet sampling. Schonfeld et al. used the last modification time of the web pages in the site metadata to select the pages to be refreshed. The web server generally stored all the URLs in the server and the last modification time as metadata. Literature[20, 21] 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. 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 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 the literature. 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. 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. proposed an accuracy-aware crawling techniques for cloud crawler that allowed local data to be re-crawled in a resource-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 speculative parallel crawler based on TLS to make full use of spare cores to make crawling programs parallelization at a lower refresh cost, and to better balance efficiency, refresh costs and freshness.

3. Thread Level Speculation-based Distributed Crawler

The parallel crawler system used in this paper adopts the structure of master-slave. 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 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 Figure 2, we can see that the main part of the framework includes master node for task generation, task allocation and scheduling, and

 doi:10.20944/preprints202410.2331.v1

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, and 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. Moreover, during the processes of page downloading, page parsing, and page optimization, Thread Level Speculation is introduced to leverage the parallelization within the multicore of one machine.

3.1. 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 the Figure 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.

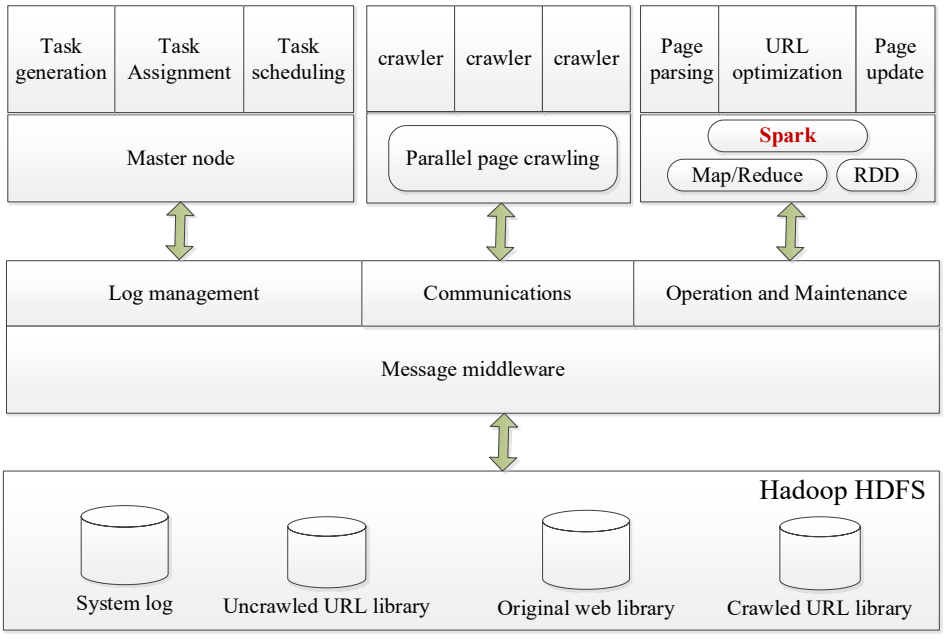


Figure 3. Distributed Parallel Crawler Framework Diagram.

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

 doi:10.20944/preprints202410.2331.v1

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.

Figure 3 shows a diagram of parallel crawler framework 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="*URLDownload*")" 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.

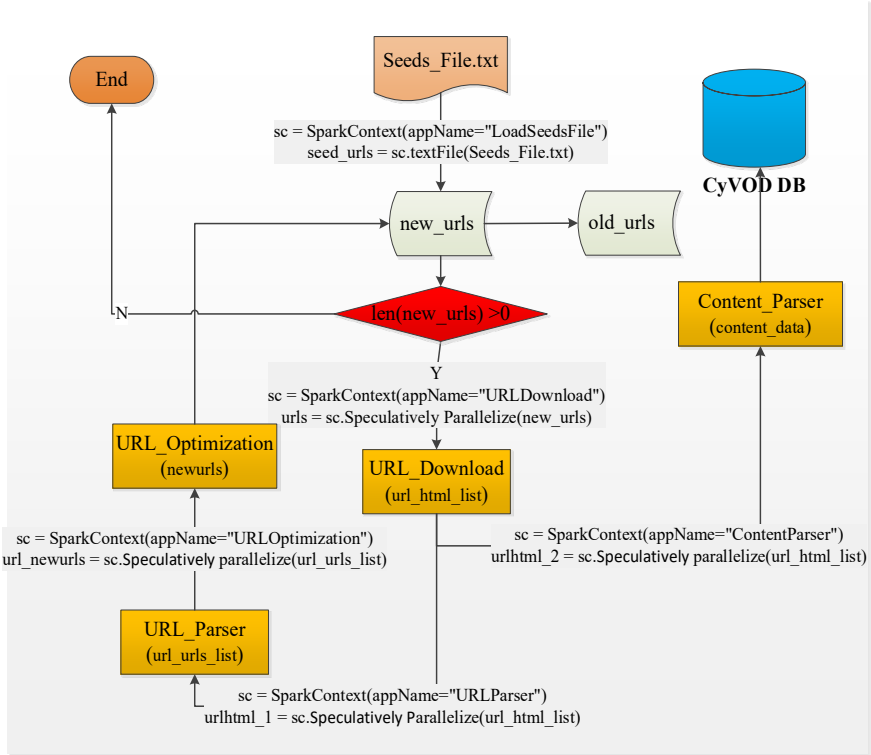


Figure 4. Parallel Crawler Framework Diagram with Spark.

3.1.1. Speculative Download with TLS

Figure 4 shows a speculatively 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 speculative parallelization. During the process of parallelization, two subprocesses need to be considered, namely in the level of cluster Spark is used to realize the parallelization among machine, while in the level of muticore speculative mechanism is applied to the parallelization among the multicores.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 30 October 2024 doi:10.20944/preprints202410.2331.v1

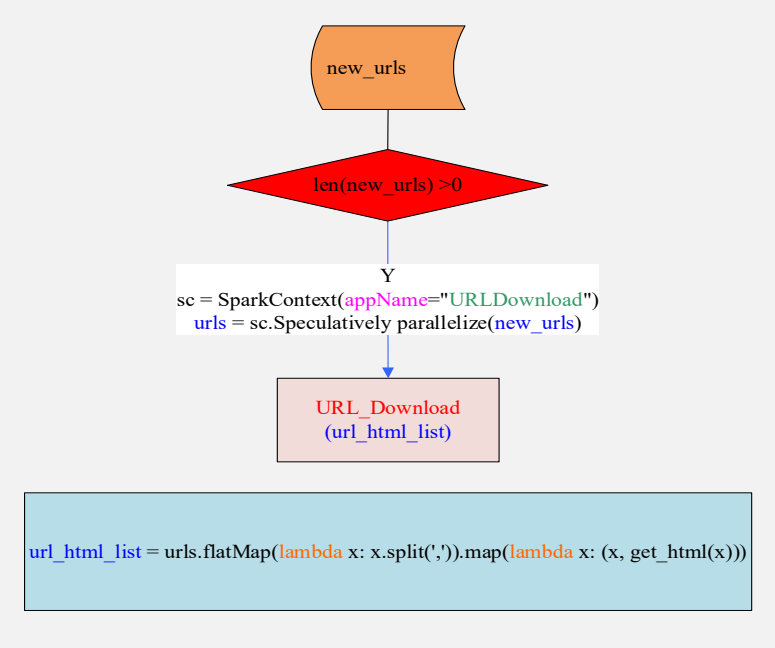


Figure 5. Parallel Download with Spark.

3.1.2. Speculative Parser with TLS

Figure 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*. During the process of parallelization, two subprocesses need to be considered, namely in the level of cluster Spark is used to realize the parallelization among machine, while in the level of muticore speculative mechanism is applied to the parallelization among the multicores.

3.1.3. Speculative Optimization with TLS

Similar to Figure 4 and Figure 5, Figure 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. During the process of parallelization, two subprocesses need to be considered, namely in the level of cluster Spark is used to realize the parallelization among machine, while in the level of muticore speculative mechanism is applied to the parallelization among the multicores.

The specific processes of speculatively 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.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 30 October 2024 doi:10.20944/preprints202410.2331.v1

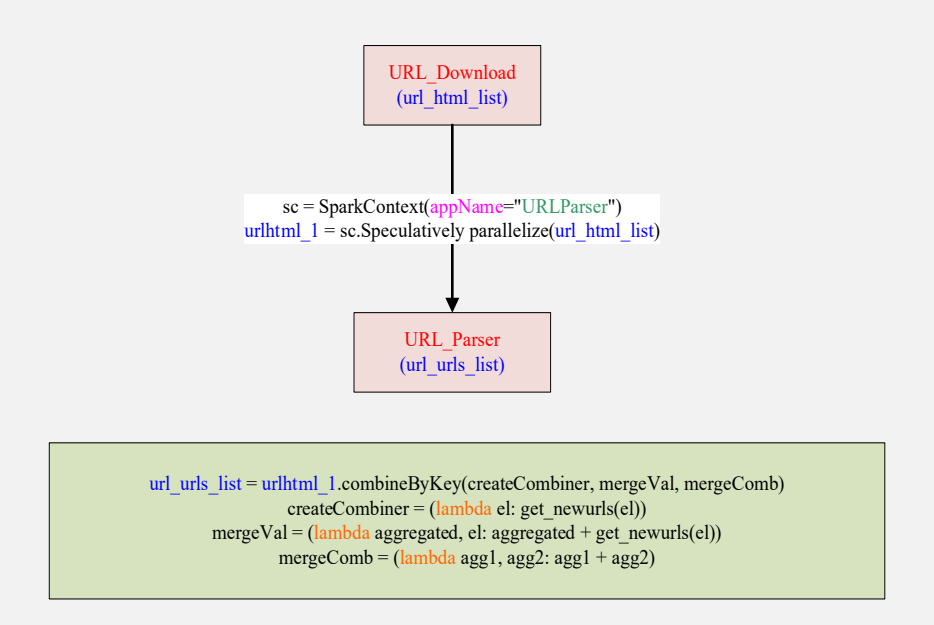


Figure 6. Parallel Parser with Spark.

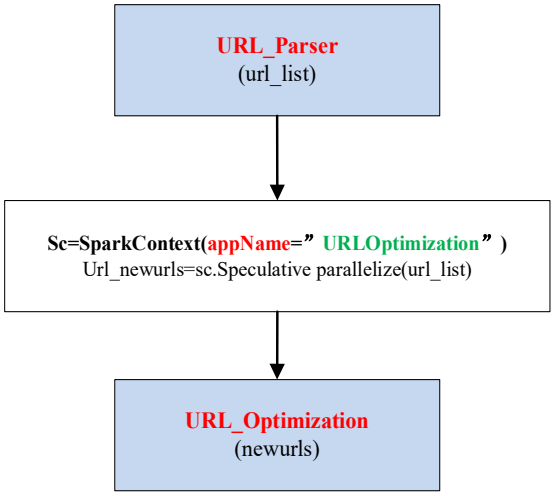


Figure 7. Parallel Optimization with Spark.

(2) Judge whether or not that the list to be fetched in the input folder is empty, if yes, skipping to (7); otherwise, executing (3).

(3) Speculatively 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) Speculatively 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) Speculatively 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 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 web page crawling plan, these pages with the crawling task at the moment are added to the crawling list.

 doi:10.20944/preprints202410.2331.v1

(9) Further parse web pages content. Analyze the content of the web pages in parallel, and then parse out the required attribute information from the merged and duplicated web pages. 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.

3.2. Parallelization Algorithm

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 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 Map/Reduce is shown in Algorithm 1.

In Table 1, 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 $part_0$~$part_{k-1}$ ($k \in$ N); Next, send $part_i$ to c_{i+1}($i \in$ N), and open a process $part_j \Rightarrow partition_j$ ($j \in$ N).

In Table 2, Algorithm 2 shows the parallel download of URL based Spark. The function u*rls_download(urls)* is defined. In the function, *SparkContext()* is used as the entry point for executing Spark applications, and the most important step in any Spark application is generating SparkContext objects. SparkContext allows Spark applications to access Spark clusters through a Resource Manager. Then, parallelization is performed by the way of map operation. Finally, *collect()* function is used to collect the final results.

In Table 3, Algorithm 3 shows the parallel parser of URL based Spark. During the process, *SparkContext()* is used as the entry point for executing Spark applications, and generating SparkContext objects is the most import step. *Get_newurls()* is used to create new *urls*, and two anonymous functions are used to merge values, *combineByKey()* is used to realize the fusion of new *urls* and old *urls*. During the process, *map()* and *reduce()* are used to perform parallelization.

In Table 4, Algorithm 4 shows the Parallel Content Parser based Spark. During the process, *SparkContext()* is used as the entry point for executing Spark applications, and generating SparkContext objects is the most import step. *Get_newurls()* is used to create new *urls*, and two anonymous functions are used to merge values, which are the parsing contents and different from the content of *get_newurls()*. *combineByKey()* is used to realize the fusion of new *urls* and old *urls*. During the process, *map()* and *reduce()* are used to perform parallelization.

In Table 5, Algorithm 5 shows the Parallel Optimization of URL based Spark. In the process, *distinct_urls()* and *url_optimi()* are used to update the *url* set through comparing the difference between old *urls* and new ones.

4. Experiment and Analysis

4.1. Experimental Configuration

This section will present a performance evaluation of SpecCA. Table 6 shows the specific configuration of experiment environment.

 doi:10.20944/preprints202410.2331.v1

4.2. Experimental Analysis

Figure 7 shows the parallel optimization with Spark, which corresponds to the Table 5. In the process, *SparkContext()* is used as the entry point for executing Spark applications, and generating SparkContext objects is the most import step. Then, speculative parallelization function-*parallelize()* is applied to realize the parallelization of *url_list*.

Figure 8 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.

Figure 9 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 right orange line represents the number of parallel crawling websites.

Figure 10 shows the changing of core number.

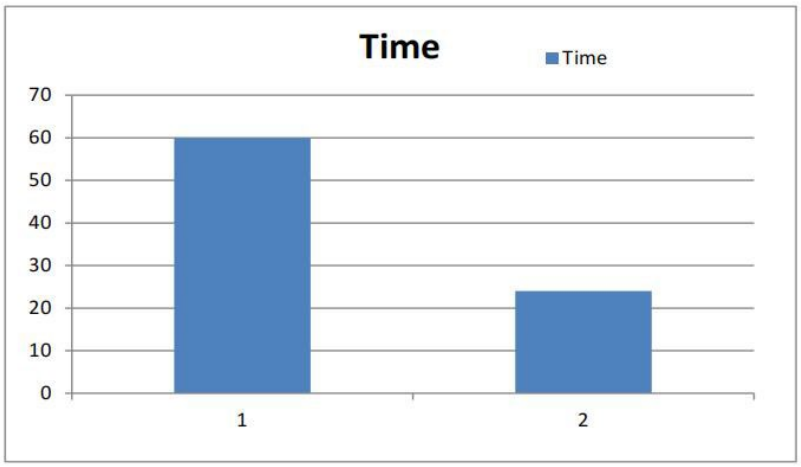


Figure 8. Spark Comparison of Crawling Time.

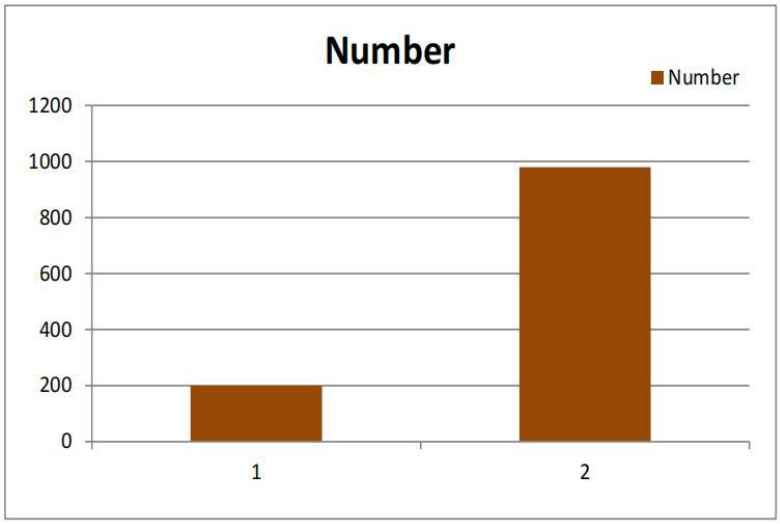


Figure 9. Comparison of Crawling Websites.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 30 October 2024 doi:10.20944/preprints202410.2331.v1

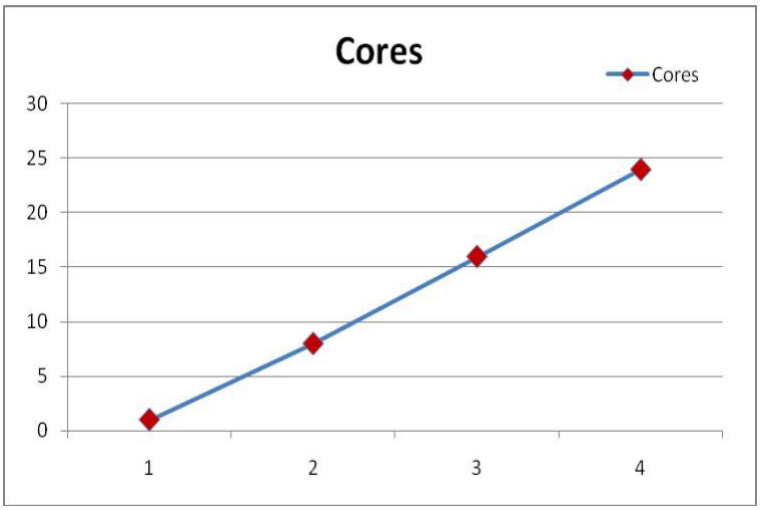


Figure 10. Number of Cores.

5. Conclusions

This paper proposes a Speculative parallel crawler approach (SpecCA) on Apache Spark. By analyzing the process of web crawler, the SpecCA firstly hires a function to divide the whole crawling process into several subprocesses which can be implemented independently and then spawns a number of threads to speculatively implement every subprocess in parallel. At last, the speculative results are merged to form the final outcome.

Table 1. 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=>{$split_0$, $split_1$, ..., $split_{m-1}$}
3:send $split_0$, $split_1$ to c_1 , open two Map processes to process this two slices
4:and send $split_2$, $split_3$ to c_2 , open two Map processes to process these two slices
5:and ....
6:and send $split_{m-2}$, $split_{m-1}$ to c_n , open two Map processes to process these two slices
7:Combine all Map output=>Inter-results
8:partitioner(Inter-results)=>{$part_0$, $part_1$, ..., $part_{k-1}$}
9:send $part_0$ to c_1 , open a Reduce process to process $part_0$ => *partition_0*
10:and send $part_1$ to c_2, open a Reduce process to process $part_1$ => *partition_1*
11:and ....
12:and send $part_{k-1}$ to c_n , open a Reduce process to process $part_{k-1}$ => $partition_{n-1}$
13:Output-File = {$partition_0$,$partition_1$,...,$partition_{n-1}$}
14: End

Table 2. Algorithm 2: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)*)

 doi:10.20944/preprints202410.2331.v1


```
    return output
```

Table 3. Algorithm 3:Parallel Parser of URL based Spark.

Algorithm 3:Parallel Parser of URL based Sp

```
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' % 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 4. 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)
```

 doi:10.20944/preprints202410.2331.v1


```
content_parser(url_html_list)
```

Table 5. 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 6. 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

 doi:10.20944/preprints202410.2331.v1

Author Contributions: Conceptualization, Yuxiang Li; methodology,Yuxiang Li; software, Yuxiang Li; validation, Yuxiang Li; formal analysis, Yuxiang Li; investigation, Yuxiang Li; resources, Yuxiang Li; data curation, Yuxiang Li; writing—original draft preparation, Yuxiang Li.; writing—review and editing, Yaning Su; visualization, Yuxiang Li; supervision, Yuxiang Li; project administration, Yuxiang Li; funding acquisition, Yuxiang Li. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Natural Science Foundation of China Grant No.61972133 and No.12101195, Project of Leading Talents in Science and Technology Innovation for Thousands of People Plan in Henan Province Grant No.204200510021, Henan Province Natural Science Fund Grant No.202300410146, Henan Province Key Scientific and Technological Projects Grant No.222102210177, No.212102210383, No.202102210162 and No.222102210072, and China Postdoctoral Science Foundation Grant No.2021M700885, and the Key Research and Development Plan Special Project of Henan Province Grant No.241111211400, and Horizontal Technology Project Grant No.22010252.

Acknowledgments: We thank our 505 and 509 laboratory for their great support during our work. We give our sincere wishes to all our colleagues of laboratory for their collaboration. We also thank reviewers for their careful comments.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

YUXIANG LI , born in 1984, earned his Bachelor,Master and Ph.D. degrees in Computer Network at Luoyang Normal College in 2008, Xi'an University of Technology in 2011 and Xi'an Jiaotong University in 2017, P. R. China.Nowadays, he is a lecturer in School of Information Engineering, Henan University of Science and Technology, P. R. China. In 2016, he, as a visitor student, focused on Thread-Level Speculation based on Machine Learning at Computer Architecture Department (CARD), in Edinburgh University, UK. Recent decade, Li made intensive efforts to pursue cutting-edge researches on computer science, including:parallel computing, machine learning, compiler optimization, and etc. His research interests primarily include parallelization of serial programs and internet of things (IoT) and deep learning. He has been engaged in investigating the parallelizing irregular programs with Thread-Level Speculation (TLS) based on machine learning on multicore processors, as well as using the theory of importance degree to assess every procedure's importance degree in one program, so to facilitate making up adaptive thread partitioning scheme for every procedure. Before proposing this idea, he was engaged in machine learning based thread partition in TLS, including generation of sample set, optimization of sample set, thread partition with machine learning methods, optimization of thread partition schemes. During this period of study, he found the shortcomings of it, namely all procedures in one program made use of one common partition scheme, making part of them can not obtain their best partitioning. As of today, in the field of TLS, Li owns 12 papers indexed by Ei Compendex database and 5 papers indexed by SCI database. Li is a CCF Member, IEEE Member. Besides, Li is the reviewers respectively for The Journal of Supercomputing (IJSC), Genomics, Proteomics&Bioinformatics (GPB), Microprocessors and Microsystems, IEEE Transactions on Parallel and Distributed Systems (TPDS).

References

1. Huang W, Peng C, Li Z, et al. AutoCrawler: A Progressive Understanding Web Agent for Web Crawler Generation[J]. 2024.
2. Liu Q, Yahyapour R, Liu H, et al. A novel combining method of dynamic and static web crawler with parallel computing[J]. Multimedia Tools and Applications. 2024, 83(21): 60343-60364.
3. Yan-Fei X U, Yuan L, Wen-Peng W U. Research and Application of Social Network Data Acquisition Technology[J]. Computer Science. 2017.
4. Wang, Hongzhi, Li, et al. Parallelizing the extraction of fresh information from online social networks[J]. Future generations computer systems: FGCS. 2016.

 doi:10.20944/preprints202410.2331.v1

5. Xia J, Wan W, Liu R, et al. Distributed Web Crawling: A Framework for Crawling of Micro-Blog Data[C]. International Conference on Smart & Sustainable City & Big Data, 2016. 2016.
6. Su L, Wang F. Web crawler model of fetching data speedily based on Hadoop distributed system[C]. 2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS), 2016. 2016.
7. Dwivedi R, Tiwari A, Jagadeesh G K. A novel apache spark-based 14-dimensional scalable feature extraction approach for the clustering of genomics data[J]. Journal of supercomputing. 2024, 80(3): 3554-3588.
8. Shoro A G, Soomro T R. Big Data Analysis: Apache Spark Perspective[J]. 2015(1).
9. Zaharia M, Xin R S, Wendell P, et al. Apache Spark: a unified engine for big data processing[J]. Communications of the Acm. 2016, 59(11): 56-65.
10. Qi R Z, Wang Z J, Li S Y. A Parallel Genetic Algorithm Based on Spark for Pairwise Test Suite Generation[J]. 计算机科学技术学报:英文版. 2016, 31(002): 417-427.
11. Qiu H, Gu R, Yuan C, et al. YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark[J]. IEEE. 2014: 1664-1671.
12. Shi K, Denny J, Amato N M. Spark PRM: Using RRTs within PRMs to efficiently explore narrow passages[C]. IEEE International Conference on Robotics & Automation, 2014. 2014.
13. Siddesh G M, Suresh K, Madhuri K Y, et al. Optimizing Crawler4j using MapReduce Programming Model[J]. Journal of The Institution of Engineers (India): Series B. 2016.
14. Dwivedi R, Tiwari A, Jagadeesh G K. A novel apache spark-based 14-dimensional scalable feature extraction approach for the clustering of genomics data[J]. Journal of supercomputing. 2024, 80(3): 3554-3588.
15. Wang Y, An H, Liu Z, et al. A Flexible Chip Multiprocessor Simulator Dedicated for Thread Level Speculation[C]. 2016 IEEE Trustcom/BigDataSE/ISPA, 2017. 2017.
16. Wang Q, Wang J, Shen L, et al. A Software-Hardware Co-designed Methodology for Efficient Thread Level Speculation[J]. IEEE. 2017.
17. Zhonghua Z, Huiran Z, Jiang X. Data crawler for Sina Weibo based on Python[J]. 计算机应用. 2014, 34(11): 3131-3134.
18. Zhi-Ming H, Xue-Weng Z, Jun C, et al. Method for Fully Distributed Crawler Cluster Based on Kademlia[J]. Computer Science.
19. Schonfeld U, Shivakumar N. Sitemaps: Above and Beyond the Crawl of Duty[J]. DBLP. 2009.
20. Meng T, Ji-Min W, Hong-Fei Y. Web Evolution and Incremental Crawling[J]. Journal of Software. 2006, 17(5): 1051-1067.
21. Calzarossa, Carla M, Tessera, et al. Modeling and predicting temporal patterns of web content changes[J]. Journal of network and computer applications. 2015.
22. Olston C, Pandey S. Recrawl scheduling based on information longevity[C]. Proceeding of the International Conference on World Wide Web, 2008. 2008.
23. Pavai G D, Geetha T V. Improving the freshness of the search engines by a probabilistic approach based incremental crawler[J]. Information Systems Frontiers. 2017, 19(5): 1-16.
24. Gupta K, Mittal V, Bishnoi B, et al. AcT: Accuracy-aware crawling techniques for cloud-crawler[J]. World Wide Web-internet & Web Information Systems. 2016, 19(1): 69-88.