Mutual Exclusion Principle for Multithreaded Web Crawlers

Kartik Kumar Perisetla
Department of Computer Science and Engineering
Lingaya's Institute of Management and Technology
Maharishi Dayanand University
Faridabad, India

***Abstract*— This paper describes mutual exclusion principle for multithreaded web crawlers. The existing web crawlers use data structures to hold frontier set in local address space. This space could be used to run more crawler threads for faster operation. All crawler threads fetch the URL to crawl from the centralized frontier. The mutual exclusion principle is used to provide access to frontier for each crawler thread in synchronized manner to avoid deadlock. The approach to utilize the waiting time on mutual exclusion lock in efficient manner has been discussed in detail.**

Keywords- Web Crawlers; Mutual Exclusion principle; Multithreading; Mutex locks.

I. INTRODUCTION

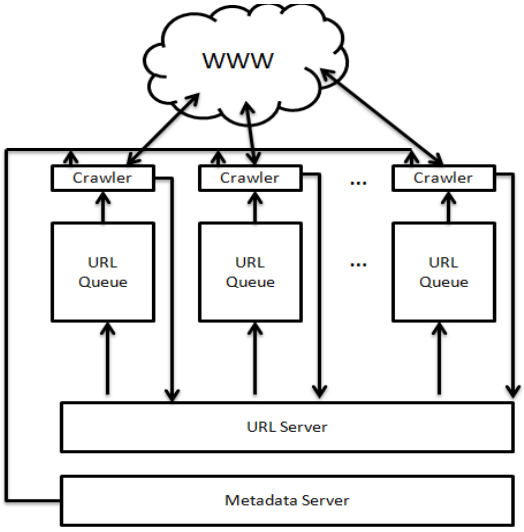
Web crawlers are programs that exploit the graph structure of the World Wide Web. The most important component of a search engine is an efficient crawler. World Wide Web is growing very rapidly; it is pertinent for search engines to opt for efficient and fast crawler processes to provide good results on search. Crawlers are also called as robots or spiders. Crawlers employed by search engines usually operate in multithreaded manner for high speed operation. When started multithreaded crawlers initialize a data structure, usually queue that holds the list of URLs to be visited by that crawler thread. These queues are filled constantly by a program employed within URL server which constantly monitors the count in each queue so that load on each crawler thread is balanced. The Load Balancing aspect is important to ensure efficient utilization of resources i.e. crawler threads. [1, 3]

Each thread start with a URL usually called a seed from their queue maintained in their local address space; they fetch the web page corresponding to that URL from World Wide Web, parse the page, extract the metadata and add links in this page to the frontier set which consists of the unvisited URLs. The data extracted consisting of body text, title, link text called as metadata are added into the metadata server. This metadata is further used by indexers for ranking the pages thus crawled. This ranked page set is then used by search engines as search results.

II. PROBLEM FORMULATION

A. Problem statement and comparison model

Traditional crawlers operate with a URL queue. The main drawback in this case is that each of them maintains a URL queue in local address. Initially, each thread holds 50 URLs to be visited. And each of them is to be monitored by single URL server program for adding new URLs to the queue as URLs are popped by crawler. Consider scenario where crawlers are operating in multithreaded manner and they access centralized URL frontier to fetch URL. Due to this, there might be cases of infinite waiting for crawler threads. To avoid such conditions and to provide synchronization among threads, mutual exclusion lock is used. Our focus is on comparison of operation model of multithreaded crawlers with synchronization lock and multithreaded crawlers without synchronization lock. We will analyze the behavior of these models and draw a conclusion based on performance.



B. Experiment model

We are considering a thread generator program capable of generating multiple crawler threads at a specified rate. Each crawler thread is capable of accessing the same centralized URL frontier, a database. The rate at which thread is generated can be easily controlled within the experimental setup to record observations. We will refer a model as "Non-mutex" when multiple threads operate without synchronization lock and we will refer a model as "Mutex" when multiple threads operate with synchronization lock to access shared resource. HTTP (Hypertext transfer protocol) is widely used for transfer of hypertext over the internet. Each thread fetches the page as

a result of HTTP request and HTTP response actions. Each web server, according to robot exclusion protocol has a file named "robot.txt" that specifies which of the pages that are changed since robots last visited. But here we are ignoring that file meant for robots. In order to indicate the benefits of mutual exclusion lock in terms of performance we have also implemented the thread generator program without the mutual exclusion lock, hence in this case it is possible for more than one crawler thread to access the URL frontier at the same time. [5]

III. Mutual Exclusion Locks For Crawler Threads

We are considering a thread generator program generates the crawler threads at a specific rate that can be tuned to different values so as to record the observations for the experiment. Mutual Exclusion principle states that multiple processes or threads intending to access the same resource will access it mutually exclusively, that is only one at a time. This can be achieved by using a binary semaphore as mutual exclusion lock, 'mutex'. Mutual exclusion for crawler threads applies in similar manner. When a crawler thread need to access the shared resource i.e. URL frontier, it check for the availability of the mutex lock. If it is in released state then it locks it and access the frontier. By that time if any other crawler threads need to access frontier it must wait until the lock is released by thread that holds the lock. Only one thread can access the URL frontier at a time hence providing controlled access and avoiding deadlock. Each thread fetches the URL to be visited from the URL frontier and establishes the connection with the web server. [6]

Pseudo code for mutex locks implementation:

```
while(mutex.isLocked())
//wait here until lock is released
Mutex.lock()
{//acquire the lock
//do processing here}
Mutex.ReleaseLock()
//release the lock
```

IV. Crawler Architecture

A. Structure

Crawler thread is the thread generated by a program. Thread runs in background mode in operating system. Crawler thread is responsible for fetching the web pages from worldwide web over HTTP. For non-mutex model, each crawler thread holds data structure for holding the raw data fetched from single source.

For mutex model, each crawler thread holds holds data structure probably a stack to hold raw data from multiple sources as discussed in latter sections. Also, in mutex model the thread generator program is responsible for providing the mutex lock to all crawler threads generated by it.

The data structure to hold the raw data is filled when HTTP response is received and it is flushed when the raw data is pushed into the raw fetched data store or the database for parsing. The threads generated by thread generator can be called as connections as each represent a connection with the web server. For example: 50 Crawler threads per sec.

B. Operation

As each thread is created it fetches a URL to be visited from the URL frontier, sends a HTTP request to the web server and waits for the HTTP response containing raw text of the page requested.

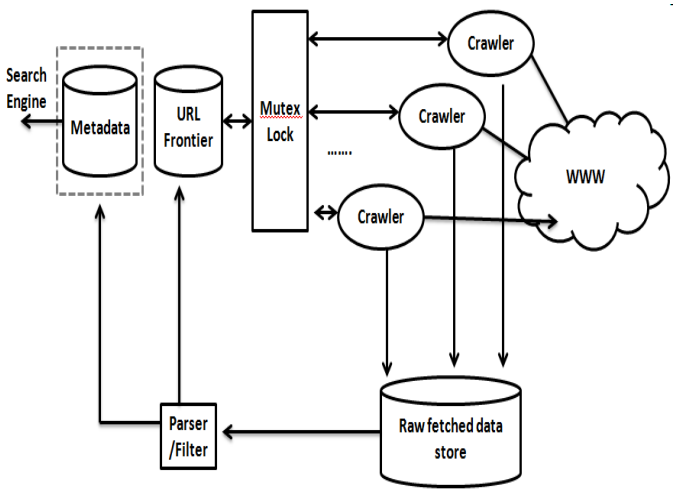


Figure 2. Mutithreaded Crawlers using the Mutual exclusion lock

By the time this thread is fetching the URL from frontier, all other threads wait for mutex lock to be released. Once the thread release the lock, another thread which was waiting for the lock acquires it. The next thread which gets this lock is dependent on how operating system manages the priority for providing the lock to next waiting thread.

The raw text thus received from HTTP response i.e. raw data is added to the 'raw fetched data store'. And then this thread repeats its action from fetching the URL. All threads will terminate when there is no URL in URL frontier. The raw data fetched is to be processed to extract metadata and links from pages. Further processing is done by the 'filter' process. It reads the page extract title, outer text of the page, link text and adds it to the metadata store. Extracts links within the page and add them to the URL frontier. [7]

C. Pseudo Code

The pseudo code for crawler thread is shown below. This gives an insight on operations performed by crawler thread and sequence of those operations.

Description of each procedure is described as:

init: This procedure is called as soon as crawler thread is created. Purpose of this method is to initialize the thread with required data structures.

fetch_url: This is responsible for fetching URL from the URL frontier by using the mutex lock.

navigate_url: This is responsible for sending HTTP request and receiving the HTTP response for a URL.

```
init( )
{ fetch_url( )
}

fetch_url( )
{ while(mutex.closed( ))
    { }
  mutex.lock( )
  new_url=pop(url_frontier)
  mutex.release( )
   If new_url is Nothing then
        {exit}
  navigate_url(new_url)
 }

navigate_url( new_url)
{send_http_request(n_url)
 get_http_response(raw)
 push(raw_data_store,raw)
 fetch_url( )
}
```

D. Crawler Algorithm

Assuming that mutex represents the mutual exclusion lock at database level that provide synchronized access to crawler threads.

1. Check the locked status of mutex lock.

   LockStatus=CheckMutexLockStatus[mutex]

2. If LockStatus=MUTEX_LOCKED then wait for lock top open by going to step 1. If LockStatus=MUTEX_OPEN then goto step 3.

3. Access the URL Frontier to pick next URL which is to be fetched and crawled to extract metadata.

   nextURL=getNextURL()

4. Release the mutex lock so that it can be accessed by other threads

   ReleaseMutexLock(mutex)

5. Fetch the raw web page and populate in appropriate data structure:

   rawData=fetchRawPage(nextURL)

6. Repeat step 1, 2 to acquire lock. Once the lock is acquired, push the rawData to database:

   pushRawPage(rawData)

7. Release the mutex lock :

   ReleaseMutexLock(mutex)

8. Repeat steps 1 to 7 until URL frontier is empty.

V. PARSER

A. Structure

Once the raw page data is pushed into the database the next step is to parse that data and extract meaningful metadata from it. This metadata acts fundamental information for search engine. The kind of elements parsed from raw data to generate metadata may vary as per the search engine requirements. In general the elements which are parsed to extract metadata are hyperlinks, title, Meta tag, headings, etc. For experiment a multithreaded parser was developed that can also generate parser threads at variable rate to extract information of raw pages and push them into database so that it can be readily used by the search engine.[8]

B. Pseudo Code

The pseudo code for Filter/Parser is shown below.

```
filter( )
{
   new_raw=pop(raw_data_store)
   new_meta=extract_meta_data(new_raw)
   push(meta_data_store,new_meta)
   extract_links(new_raw )
}
extract_meta_data(new_raw)
{
      //Extracts and returns the Metadata of the
page
}
extract_links(new_raw)
{
    for each url in new_raw
    {
       push(url_frontier,url)
     }
}
```

filter: This procedure is called as soon as filter process is initiated. Purpose of this method is to initialize the thread with required data structures.

extract_meta_data(new_raw): This procedure is responsible for extracting meta data from the page and adding it to raw fetched meta data store.

extract_links(new_raw): This procedure is responsible for extracting all URLs from the page and add them to URL frontier.

C. Parser Algorithm

Assuming that mutex represents the mutual exclusion lock at database level that provide synchronized access to parser threads.

1. Check the locked status of mutex lock.

   LockStatus=CheckMutexLockStatus[mutex]

2. If LockStatus=MUTEX_LOCKED then wait for lock top open by going to step 1. If LockStatus=MUTEX_OPEN then goto step 3.
3. Access the raw page data from database:

   rawData=GetRawPageData()
4. Release the mutex lock so that it can be accessed by other threads

   ReleaseMutexLock(mutex)
5. Parse the page and extract metadata from it:

   metaData=ExtractMeta(rawData)
6. Repeat step 1, 2 to acquire lock. Once the lock is acquired, push the metaData to database:

   pushMetadata(metaData)
7. Release the mutex lock :

   ReleaseMutexLock(mutex)
8. Repeat steps 1 to 7 until URL frontier is empty

VI. OBSERVATIONS

A. Time factor

Consider let T be the combined time to fetch a page from the web, extract metadata and links from it. Now this T is composed of two components: time to fetch the page from web and time to parse the web page to extract links and metadata. Let t_f be the time to fetch the page and t_p is the time to parse the page to extract data from it. Then we can write T as:

$$T = t_f + t_p$$

A set of 2000 URLs is serving as the URL frontier at the beginning of the experiment. We performed our experiment for both crawling using mutex lock and crawling without mutex lock. Crawler threads are only responsible for fetching the web pages not parsing the pages. A 'Filter' program is used to parse the fetched web pages, extract links and metadata from them. Since we are using same URL frontier set for both mutex based and non-mutex based crawling, the 'Filter' program takes same constant amount of time to parse pages for both the cases. t_f includes the time to fetch the URL from frontier, time to send HTTP request and time to obtain the HTTP response. t_f can be written as:

$$t_f = t_{request} + t_{response}$$

Where $t_{request}$ is the time taken by request to reach the server and $t_{response}$ is the time taken for response to reach the crawler. Above equation holds good for models where the parser can directly get the raw data from the crawler thread for parsing. For models where the parser threads write the raw page data fetched from a URL to the centralized database, the equation can be written as:

$$t_f = t_{request} + t_{response} + t_{pushToStore}$$

$t_{pushToStore}$ is the time to acquire the mutex lock, write the raw data and to release the lock. $t_{request}$ can be further broken down into $t_{pickurl}$ and $t_{httprequest}$. $t_{pickurl}$ is time spent waiting for mutex lock, acquire mutex lock for database, access next URL and release the mutex lock. $t_{httprequest}$ is the time taken to create HTTP request and send it to respective endpoint. $t_{response}$ depends on several factors like speed of the internet connection, load on the web server serving that page and many other factors. The only parameter we can control is $t_{request}$. This is the only factor that can be controlled to minimize the t_f.

B. Time Minimization

The minimization of $t_{request}$ was performed in this experiment within the variable rate crawler thread generator. Generator provides provision to set rate at which the crawler thread will be generated. Once the page is crawled, its raw source is pushed into database with other relevant information specific to URL resource. The parser threads are responsible for parsing the raw page and extract useful metadata from it that can be fed to the search engine. These threads too are executed in multithreaded manner where synchronization between thread is done through mutex lock at database level. Based on observations recorded by generating crawler threads at variable rates, a graph is plotted for $t_{pickurl}$ against threading rate and is shown below:

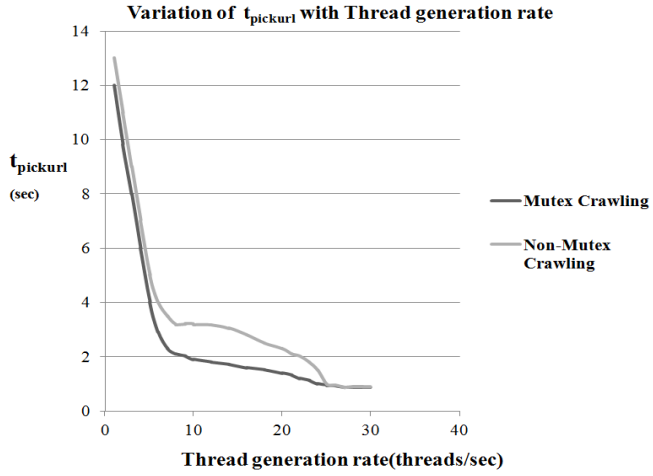


Figure 3. Graph for $t_{pickurl}$ vs. thread generation rate

C. Utilizing mutex lock waiting time in crawler thread

Consider the case when a crawler thread holds the mutex lock and other threads are waiting for the lock to read the next URL from the frontier. Here we are considering the mutex model where mutex lock is used for synchronization. Under normal operation conditions the probability of majority of threads waiting for mutex lock is high. This totally depends on the t_f, the time to fetch the raw page. It was observed that majority of threads have similar t_f. Thus they end up fetching the page in same time and spend most of time waiting for mutex lock to fetch next URL. The waiting time for crawler thread can be utilized by employing that time for fetching raw data for subsequent URLs. We name this approach as extended crawling. The change required in crawler thread is that rather than picking a single URL from the frontier it picks collection of URLs whose raw data is to be fetched. This collection of URLs is pushed onto a stack STK[URL]. Once raw page data for a URL is fetched crawler checks for availability of mutex lock. If lock is held by any other thread

then current thread pushes the raw fetched data onto a stack STK[RAW] and pops the next URL from the STK[URL]. Then crawler fetches the raw page data for this next URL popped. So this way the waiting time is utilized for fetching raw data for collection of URLs. In this model each thread will push the raw data to database in short bursts whenever the mutex lock is acquired by the thread.

The proposed algorithm for utilizing the waiting time for extended crawling can be written as:

1. Check the locked status of mutex lock
   LockStatus=CheckMutexLockStatus[mutex]

2. If LockStatus=MUTEX_LOCKED then goto step3. If LockStatus=MUTEX_OPEN then goto step 7

3. Pop the URL from STK[URL] to fetch the page while the mutex lock is held by other threads:
   nextURL=STL[URL].Pop()

4. Fetch the raw page data using the URL popped in previous step:
   rawData=HTTPFetch(nextURL)

5. Push the fetched raw data onto STK[RAW]:
   STK[RAW].Push(rawData)

6. Repeat Step 1 to acquire the lock.

7. Pop the fetched raw page data form top of stack STK[RAW] and write it to database:
   rawData==STK[RAW].Pop()
   CommitToDatabase(rawData)

8. Repeat step 7 until stack is empty. Once stack is empty repeat steps 1 to 6.

Consider the variation of t_f, $t_{request} + t_{response}$ and $t_{pushToStore}$ with thread generation rate for a single thread. The dark shaded region shows the time spent in sending the request and fetching the page in the waiting time for the mutex lock by crawler thread. The dark black line shows the variation of total time to fetch the page (t_f). The light shaded region shows the variation of $t_{pushToStore}$ with thread generation rate. In case the mutex waiting time would not have utilized, the region under dark line (t_f) will be light shaded which mainly consists of time spent waiting on lock after the page is fetched. The following graph shows that large portion of t_f, i.e. waiting time on mutex lock is utilized for fetching raw pages for subsequent URLs.

D. Utilizing mutex lock waiting time in parser thread

Consider t_p, this factor is highly variable based on the amount of elements on crawled page. Higher the number of elements on the crawled page, higher the parsing time. t_p can be broken down into t_{parse} and $t_{pushMetadata}$. t_{parse} is the time taken to parse the raw page and fill the appropriate data structures.

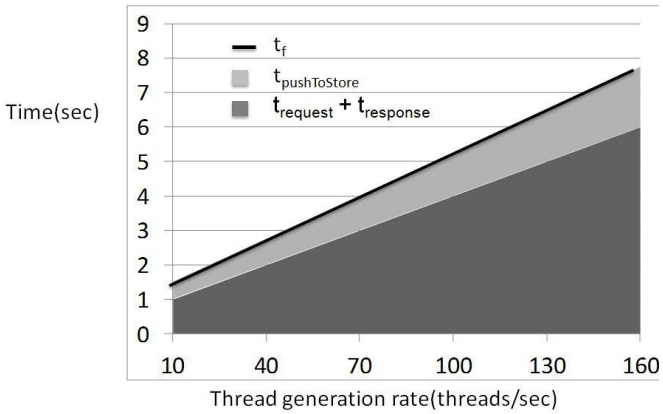


Figure 4. Graph for t_p vs. thread generation rate

The table shows the observations for $t_{request} + t_{response}$ and $t_{pushToStore}$ at different number of crawler threads:

TABLE I. Variation of $t_{request} + t_{response}$ and $t_{pushToStore}$ with thread generation rate

Parser Threads	$t_{request} + t_{response}$ ***(sec)***	$t_{pushToStore}$ ***(sec)***
10	1	0.5
40	2	0.75
70	3	1
100	4	1.25
130	5.1	1.5
160	6	1.75

$t_{pushMetadata}$ is the time spent waiting for mutex lock, acquire mutex lock, save changes in database, commit the changes and release the mutex lock. Under normal operation conditions the probability of majority of threads waiting for mutex lock is high. The reason is that most of threads might finish parsing operation at same time and they wait for lock if it is acquired by other thread. The waiting time for a parser thread can be utilized by employing that time for parsing subsequent raw pages by picking up another raw page data and parsing it. We name this approach as extended parsing.

The change required in parser thread will be that rather than fetching raw page data for single page the parser will fetch collection of raw page data from the database and push collection onto a stack STK[RAW]. Once the parser finishes parsing raw page and if the mutex is locked then parser can pop raw page data for other pages held in stack and start parsing them. The parsed metadata set can be pushed on the stack STK[META] for pages parsed while waiting for mutex lock. Once the lock is acquired by the thread, it can write all parsed metadata which is held on stack to the database and release the lock. In this model each thread will push parsed metadata to database in short bursts whnever the mutex lock is acquired. This way the waiting time for mutex lock can be utilized for parsing the raw page.

The proposed algorithm for utilizing the waiting time for extended parsing can be written as:

1) Check the locked status of mutex lock

2) LockStatus=CheckMutexLockStatus[mutex]

3) If LockStatus=MUTEX_LOCKED then goto step3. If LockStatus=MUTEX_OPEN then goto step 7.

4) Pop the raw page data from STK[RAW] to extract metadata from the page while the mutex lock is held by other threads:

5) rawData=STL[RAW].Pop()

6) Parse the page and extract metadata from it:

7) metaData=ExtractMeta(rawData)

8) Push the extracted metaData onto STK[META]:

9) STK[META].Push(metadata)

10) Repeat step 1 to acquire the lock.

11) Pop the metadata from top of stack STK[META] and write the metadata to database:

12) metadata= STK[META].Pop()

13) CommitToDatabase(metadata)

14) Repeat step 7 until stack is empty. Once stack is empty repeat steps 1 to 6.

Consider the variation of t_{parse}, $t_{pushMetadata}$ and t_p with thread generation rate for a single thread. The dark shaded region shows the time spent in parsing the raw pages in the waiting time for the mutex lock by filter thread. The dark black line shows the variation of total time for parsing (t_p). The light shaded region shows the variation of $t_{pushMetadata}$ with thread generation rate. In case the mutex waiting time would not have utilized, the region under dark line (t_p) will be light shaded which mainly consists of time spent waiting on lock after parsing is complete. The following graph shows that large portion of t_p, i.e. waiting time on mutex lock is utilized under the parsing for subsequent set of raw pages.

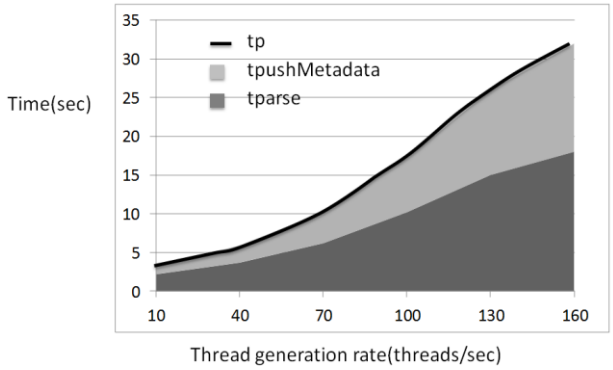


Figure 5. Graph for t_p vs. thread generation rate

The table shows the observations for t_{parse} and $t_{pushMetadata}$ at different number of parser threads:

TABLE II. Variation of t_{parse} and $t_{pushMetadata}$ with thread generation rate

Parser Threads	t_{parse} *(sec)*	$t_{pushMetadata}$ *(sec)*
10	2.2	1
40	3.7	2.2
70	6.2	4
100	10.2	7
130	15	11
160	18	14

VII. Results

The experiment was conducted on Windows XP sp-2 operating system equipped with 512MB RAM, 512 kbps ADSL broadband connection. We are calculating time t_f, the time to fetch the fetch the page. The variation of $t_{pickurl}$ with thread generation rate has been discussed. Also, the experiment results involving utilization of mutex waiting time for parsing raw pages indicates the gravity of the approach. It can be deduced from the graph that multithreaded crawlers works efficiently only with the usage of mutual exclusion lock.

We can observe that for lower rate values, small increase in rate brings down $t_{pickurl}$ by large amounts. For larger rate values, large increase in rate brings small change in $t_{pickurl}$.

Also, it presents new approach to utilize mutex waiting time for parsing operation. This leads to increased performance of the crawler, parser and efficient utilization of resources.

VIII. Future Scope

The future work will focus on minimizing the time incurred in acquiring the lock, writing data to database and releasing the lock. This time is represented as grey section in graphs shown in this document.

This may be accomplished by interacting with operating systems at a lower level to speed up the locking and releasing the mutex lock. Also, we will cover the aspects that will enhance the performance by providing an efficient synchronization model across crawler and parser threads.

IX. Conclusion

This paper presented a new approach for implementing multithreaded crawlers using mutual exclusion locks, which results in performance improvement as compared to traditional crawlers.

The approach of utilizing mutex waiting time proves efficient if employed for parsing or other useful operations within crawler threads.

References

[1] Lawrence Page, Sergey Brin. The Anatomy of a search Engine. Submitted to the Seventh International World Wide Web Conference (WWW98). Brisbane, Australia

[2] Budi Yuwono, Savio L.Lam, Jerry H. Ying, Dik L. Lee. A World Wide Web Resource Discovery System. The Fourth International WWW Conference Boston, USA, December 11-14, 1995.

[3] Gautam Pant, Padmini Srinivasan, Filippo Menczer. Crawling the Web.

[4] Allan Heydon, Marc Najork. Mercator: A Scalable, Extensible Web Crawler

[5] Muhammad Shoaib, Shazia Arshad. Design and Implementation of web information gathering system

[6] Joo Yong Lee, Sang Ho Lee. Scrawler: A Seed by Seed Parallel Web Crawler.

[7] Boldi P., Codenotti B., Santini M., and Vigna S. UbiCrawler: a scalable fully distributed web crawler. Software Pract. Exper., 34(8):711–726, 2004

[8] S.chakraborti, M.van den Crawling: A new approach to topic-specific web resource discovery". In the 8th International World Wide Web Conference, 1999

AUTHORS PROFILE

Kartik Kumar Perisetla received his Bachelors degree in Computer Science from Lingaya's Institute of Management and Technology. He is currently working as Software Engineer. His research interest include Grid Computing, Machine Learning and Web Crawling