Techniques To Implement Web Crawlers Using Multi-Threading

Shruthi Ramprasad
Computer Science and Engineering
Santa Clara University
Santa Clara, US.
sramprasad@scu.edu

Divya Mahajan
Computer Science and Engineering
Santa Clara University
Santa Clara, US.
dmahajan2@scu.edu

Xi Liu
Computer Science and Engineering
Santa Clara University
Santa Clara, US.
xliu11@scu.edu

Abstract*—This paper aims to implement multithreaded web crawlers with and without locks. There is a shared resource for multiple web crawlers, which is the URL frontier queue. The URL frontier queue is of fixed length, which stores the URLs that need to be crawled. Each crawler is going to extract a task (URL to crawl) from the frontier, crawls this web page to find further links available on this page, and push the fetched links into the frontier for further crawling. To avoid collision inside the URL frontier, locks must be employed for any push or pop from the frontier queue. This paper distinguishes the optimistic locking system and pessimistic locking system and implements two locking mechanisms: semaphore and monitor. It also analyzes the Coarse-grained locking technique and Fine-grained locking technique for the web crawlers. The performance of the different locking techniques and lock-free approach can be evaluated by the number of threads crawled within a given time slice.

***Keywords*— multithreaded web crawlers, semaphore, monitor, performance**

I. INTRODUCTION

A web crawler, also called a robot or spider, simulates a browser to send a request to the server, traverses through the hyperlinks in web pages, and downloads web documents in an automated manner to facilitate search engines. A single thread web crawler starts with a seed URL, extracts a web page of that URL from the World Wide Web, and then parses the raw data crawled from that URL, adds unvisited links to the URL frontier for further crawling and sends metadata to the database, which will be indexed and used as a search result.[2]

The significant time consumer of a web crawler is the I/O block time when requesting web pages and waiting for the response. Therefore, to enable multi-threading and allow waiting of different requests at the same time can greatly improve the crawling efficiency. In this scenario, the safety and liveliness in the critical section of the system are under consideration. The architecture of the multithreaded web crawlers is each crawler takes a URL from the central URL frontier to work on and pushes back new links it has fetched into the frontier for further crawling. So, two crawlers, with optimistic locking technique, may extract the same URL or overwrite each other's fetched links into a fixed size frontier. If trying to prohibit concurrent conflicting acts each time, it is not ideal for efficiency. Pessimistic locking technique is employed to solve the problem with the lock-free model. This project applies semaphore and monitor locking mechanisms to the central URL frontier respectively. The locks guarantee that only one thread can access the URL frontier and change it each time, thus no two crawlers fetch the same URL to crawl and the URLs in the frontier cannot be overwritten. After being allocated the URL tasks, multiple crawlers then can send request and do their crawling job simultaneously. The total number of pages crawlers have crawled is the parameter to evaluate the efficiency.

II. BACKGROUND

The research paper [1] provides details on some of the crucial elements of a multi-threaded web crawler and some of the algorithmic particulars. The paper describes the parallelization of the crawler and how the crawler can be used to download the web pages. Also, it considers the algorithmic particulars and some of the essential elements in a web crawler. This paper emphasizes the multiple independent, communicating web crawler processes called as the C-procs. It uses C-Proc to download the document.
The second research paper [2] explains how mutual exclusion principle be achieved in a multi-threaded web crawler system and how each crawler thread in a multi-threaded system can be used to access the frontier queue in a synchronized manner to avoid the potential deadlock situation and achieve higher performance. To achieve mutual exclusion and synchronization among the threads, binary semaphore such as the mutual exclusion lock "mutex" is used. Whenever the crawler is fetching the URLs from the frontier it checks for the availability of the mutex lock. Once the lock is acquired the data is fetched from the URL Frontier. During this time, all the other threads

which need to fetch the URL from the Frontier wait for the mutex lock to be released. This paper also compares Mutex Crawling v/s Non mutex Crawling and analyzes the performance by plotting the graph for the thread generation rate and time taken to pick URL. It also analyzes the time taken to fetch the page and states that the multi-threaded crawlers work efficiently only with the usage of the mutual exclusion locks.

Further, the research paper [4] explains the fine-grained locking mechanism and how it can automate to achieve synchronization. In this paper, the classic reader-writer problem strategy is used between the locks and the shared objects and assumes the locks to be reentrant. Also, this research paper [4] proposes an algorithm to automatically insert the fine-grained locking system into a tree shaped shared data structure which acts as a critical section and proves that this algorithm ensures serializability and deadlock freedom.

Furthermore, the research paper [5] provides details on the methods to detect the change frequency of web pages efficiently in a Multi-threaded Web Crawler system. The experiments conducted in this paper highlight the importance of the usage of the multi-threaded crawler system and optimization of the change detection process which can be used in a high-performance server. The experiments conducted in this paper highlight that as the number of threads used for the web crawling increases, the effective time taken for the change detection gets reduced.

As mentioned above, a wide range of research has been carried out to use the multi-threads to increase the performance of the multi-threaded web crawler system and various locking mechanisms are used to achieve mutual exclusion and performance analysis of the web crawler systems w.r.t the single thread and multi-threads are carried out.

III. DESIGN AND ARCHITECTURE

A. *Optimistic*

The locking mechanism is a popular concurrency control mechanism in distributed computing systems and is used to ensure data integrity by prohibiting concurrent conflicting updates on shared data objects. It was first proposed by H. T. Kung and John T. Robinson[12][13].

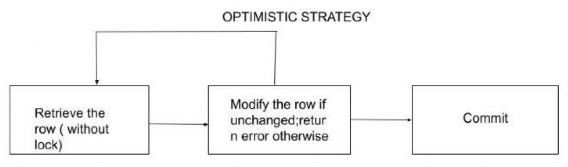

OPTIMISTIC STRATEGY
Retrieve the row (without lock)
Modify the row if unchanged;retur n error otherwise
Commit

Figure 1 Optimistic strategy

This method assumes that multiple transactions can frequently be completed without interfering. It also assumes that conflicts do not occur during execution time. Whenever the transaction is executed, no synchronization will be performed. However, at the end of each transaction, a check is performed to ensure there are no conflicts occurring in the executed transaction. If the check performed detects a conflict, the transaction will be aborted. Otherwise, the transaction is committed. Since conflicts do not occur very often, this algorithm is very efficient compared to other locking algorithms.

The running transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. However, if the check conducted reveals any modifications on the data resources, the committing transaction is rolled back. And this can be restarted again. If there are multiple changes made on the committing transactions, it would be redundant to perform the check, rollback, and restart operations on the system repeatedly. Optimistic concurrency control is used in environments with low data contention. When conflicts are rare, transactions can be completed without the expense of managing locks and without having transactions wait for other transactions' locks to clear.

B. *Pessimistic*

Pessimistic locking is also known as the record locking. It is the technique of preventing simultaneous access to the shared data to prevent inconsistent results [14][13].

In this method, the lock will be used to protect the concurrent update on the shared data. The lock is placed on the shared data, whenever the data is being modified by any user, so that no other user can use this data. This prevents records from being overwritten incorrectly but allows only one record to be processed at a time, locking out other users who need to edit/modify records at the same time.

To allow several users to edit the shared data/record simultaneously and prevent inconsistencies created by unrestricted access, a lock can be used on the single record for the modification of any record values of the shared data resources. Anyone attempting to retrieve the same record for editing is denied write access because of the lock (although, depending on the implementation, they may be able to view the record without editing it). Once the record is saved or edits are canceled, the lock is released. Records can never be saved to overwrite other changes, preserving data integrity.

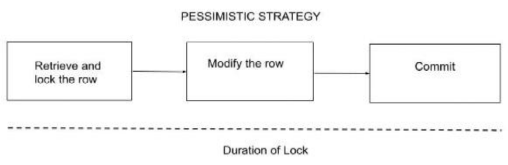


Figure 2 Pessimistic strategy

The semaphore and monitor locking architectures in this paper are based on the idea of pessimistic locking technique.

C. Coarse-Grained

The coarse-grained locking mechanism locks the complete data structure, before performing any operation [10]. In this paper, the frontier queue is now converted into a linked list. This list itself has a single lock which every procedure must acquire before the call. This makes sure that all the actions performed on the list are by holding the lock. This ensures sequential execution.

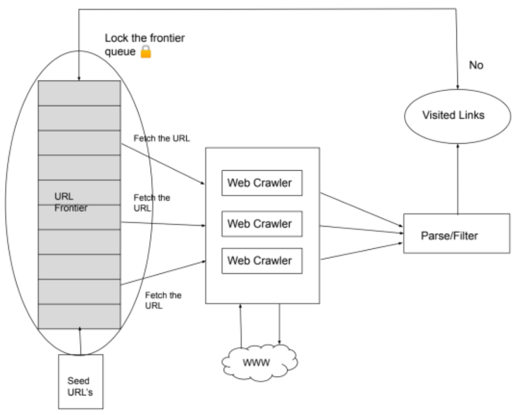


Figure 3 Architecture of synchronization of multithreaded web crawlers using coarse-grained locking

For insertion, lock at the beginning of the insertion method and insert the URL at the front of the linked list naming it the new head.

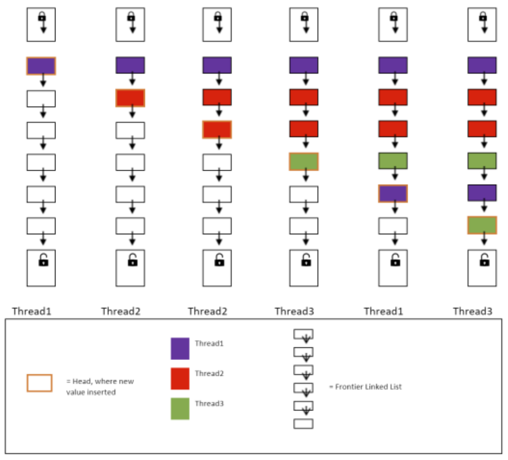


Figure 4 Coarse-grained multithreaded synchronization insertion of URL in web crawler frontier [10]

For deletion, lock at the beginning of the removal method and remove the head of the linked list, i.e., the URL. The new head is the next node of the current head.

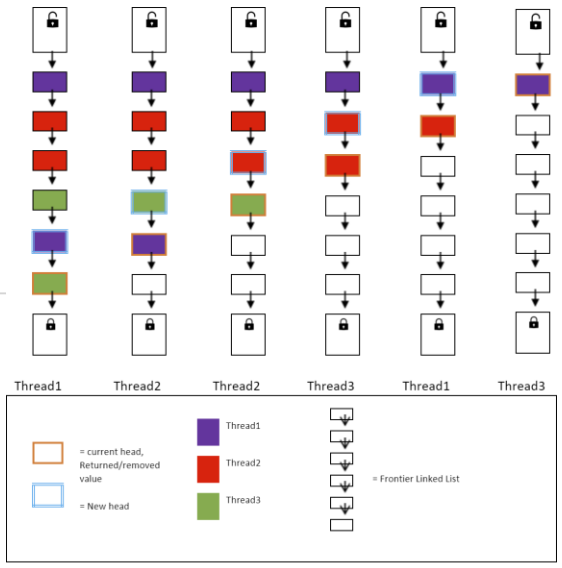


Figure 5 Coarse-grained multithreaded synchronization removal of URL in web crawler frontier [10]

D. Fine-Grained

The fine-grained locking approach locks a minimal subset of the data structure. As such, operations on different data index may continue simultaneously unimpeded. This helps to achieve greater concurrency and often better performance. Rather than protecting all system resources, each fine-grained lock will protect a single resource or a small subset of them.

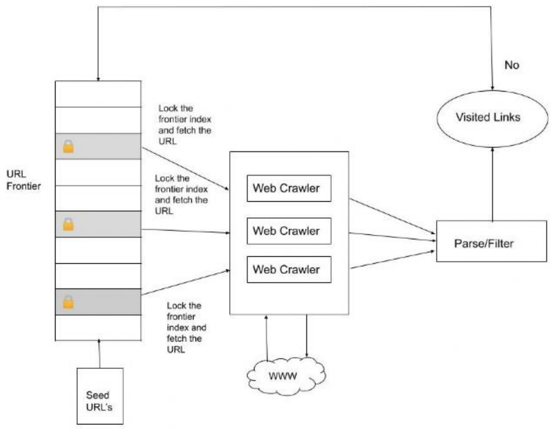

No
Visited Links
Lock the frontier index and fetch the URL
Lock the frontier index and fetch the URL
Lock the frontier index and fetch the URL
URL Frontier
Web Crawler
Web Crawler
Web Crawler
Parse/Filter
WWW
Seed URL's

Figure 6 Architecture of synchronization of multithreaded web crawlers using fine-grained locking

Rather than holding a lock for a long time (as coarse-grained locks, where a process holds the whole-system lock for as long as it runs), each process will hold this lock for as little time as possible while still providing protection. The advantage of this approach is concurrency (and therefore performance). In a fine-grained locking system, processes aim to hold minimal sets of locks and aim to achieve mutual exclusion along with parallelism in the operations, involving the shared data operations.

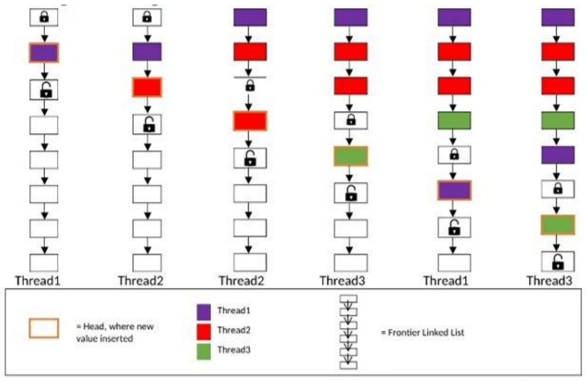


Figure 7 Fine-grained multithreaded synchronization insertion of URL in web crawler frontier [10]

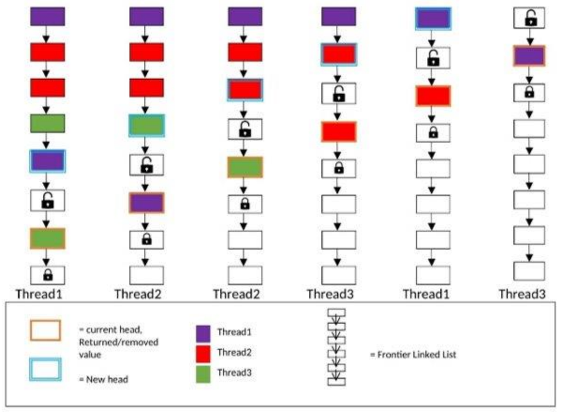


Figure 8 Fine-grained multithreaded synchronization removal of URL in web crawler frontier [10]

E. Lock-Free [1]

In traditional multi-threaded programming environment, locks are used to synchronize the access to the shared resources and to prevent race conditions. If one of the threads attempts to access a critical region that is locked by another thread, the thread would be blocked until the currently lock acquired thread releases the lock on the shared resources. It is not always desirable to block a thread because, a blocking thread might be performing a high priority or real time task and it would be inefficacious to halt its progress/put the thread into the blocked state. However, the use of locks can sometimes lead to problems such as deadlock, livelock and priority inversion. The non-blocking algorithms do not face these downsides. A lock-free data structure can be used in such scenarios to avoid the problems caused by thread blockages and improve the performance.

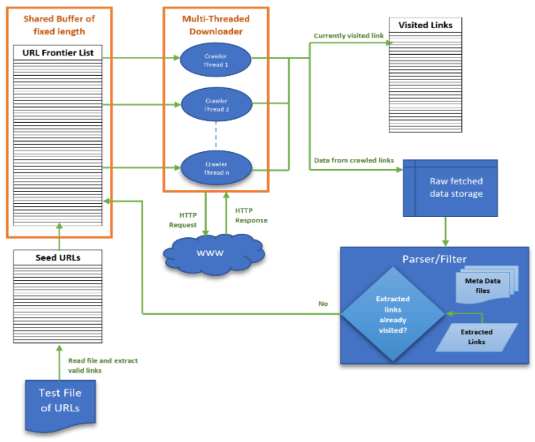


Figure 9 Architecture of lock-free parallel web crawlers [1]

Figure 9 represents the lock-free architecture of the multithreaded web crawlers. The frontier queue is the fixed buffer. The input file with seed URLs and the parser which extracts links from crawled URLs populate the frontier queue. When the crawler is executed, the URLs are assigned to the crawler threads. A lock-free data structure improves parallel performance because of the access to the shared data resources. No thread can block the other threads access to the critical section. An algorithm can be classified into the lock-free algorithm when the threads in the program run for sufficiently longer time and at least one of the threads can make progress and run to completion. However, collision can happen in the shared area, that is two parallel threads can fetch the same URL to crawl or overwrite the extracted URLs into the frontier queue.

F. *Semaphore*

Semaphore is a locking technique first introduced in 1965, by E. W. Dijkstra. There is a positive counter in each semaphore, and the value of the counter means the number of threads waiting to be wakened up. There are two functions for semaphore, which are the P function and the V function. P function is to decrement the counter value by 1 and wait while the counter is not positive. V function is to increment the counter value and wake up waiting threads.[7]

This paper applies three semaphores for the fixed size URL frontier. One semaphore is to count the total full slots with URL in the frontier. When there is at least one full slot existing, the web crawler is allowed to access the frontier to fetch the URL. The second semaphore is to count the total empty slots in the frontier. Crawlers are only allowed to input new links into the frontier when there is an empty slot to avoid overwriting the frontier. The last semaphore is a binary semaphore, which means the counter value can only be 0 or 1. This semaphore is to implement the mutual exclusion for the frontier.

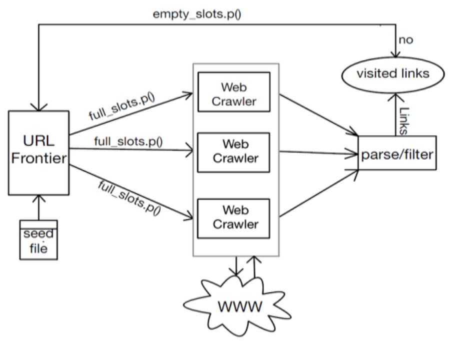


Figure 10 Architecture of synchronization of multithreaded web crawlers using Semaphore [1]

G. *Monitor*

Brinch Hansen (1973) and Hoare (1974) proposed a higher-level synchronization primitive called a monitor to reduce the complexity of synchronizations using semaphore and their orders, which may lead to deadlock. [7] This experiment has taken the analogy of the Producer-Consumer [7] problem in the multithreaded web crawlers and tried to resolve it by Monitor synchronization. The frontier queue is the fixed buffer. The input file with seed URLs and the parser who extracts links from crawled URLs are the producers. While crawler threads are consumers of the URLs from the frontier.

In this paper, the critical section i.e., the frontier queue is encapsulated inside the monitor. The monitor is using a collection of procedures to insert and remove URLs from the frontier queue. The frontier queue itself is a private object to monitor. And to insert a new URL or to get a URL from this frontier, the procedures are called inside the monitor. To achieve mutual exclusion, only one action upon the critical region (frontier queue), which is either the insertion or removal procedure, can be active at any instant of the monitor. The overlapping instructions to call the procedure are suspended.

Condition variables along with operation wait ensure to block the insertion thread while inserting in a full frontier. The other operation signal of the condition variable is used to awaken the blocked insertion thread, whenever a URL is removed from the frontier by another thread. These operations are simultaneous in the case when a thread attempts to pull a URL from an empty frontier.

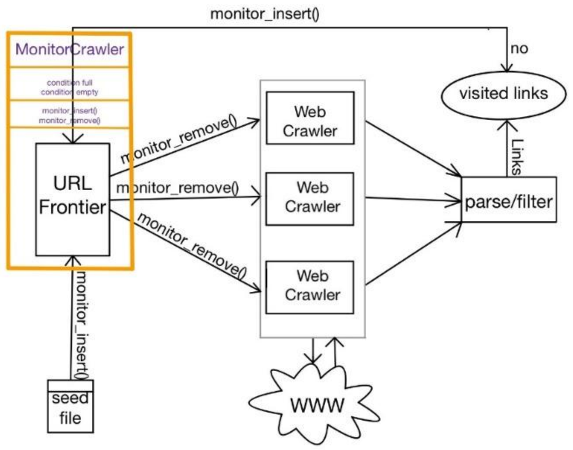


Figure 11 Architecture of synchronization of multithreaded web crawlers using Monitors

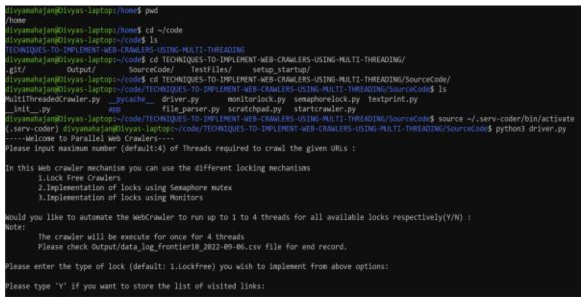
IV. EXPERIMENTAL STUDY

A. *Implementation*

Initially, the project aims to understand and implement the multithreaded web crawlers with different locking techniques and to analyze its performance. The crawler threads share a common fixed size frontier queue. The user input is validated and the valid URL's are stored in the seed URL list and is inserted to the frontier queue. In such a scenario, it is necessary to ensure that the crawler thread does not read the same URL and perform the work of scraping the pages. Also, it is a challenging task to ensure that the URL's in the frontier queue are not overwritten by the input seed URL's. Further, in our experiment, to analyze the performance of the designed web crawlers using different locking mechanisms, frontier queue size is fixed, and the max running time of the program is set to 3 seconds. These conditions were included to ensure and analyze the number of links crawled in the given max running time with respect to the number of threads. It helps to analyze the optimistic locking and pessimistic locking for the crawling tasks. Hence the below mentioned locking mechanisms were simulated during the experiment study.

(1) Lock-free
(2) Semaphore
(3) Monitor

We have utilized the python libraries to implement and analyze the above-mentioned locking mechanism. The experiment the implementation of the multithreaded web crawler system is conducted using the python3 programming language. Furthermore, these experiments were carried out on a system with 10 cores(8 Performance 2 efficiency), 32 GB ram size and 1TB secondary memory.



```
divyamahajan@Divyas-laptop:/home$ pwd
/home
divyamahajan@Divyas-laptop:/home$ cd ~/code
divyamahajan@Divyas-laptop:~/code$ ls
TECHNIQUES-TO-IMPLEMENT-WEB-CRAWLERS-USING-MULTI-THREADING
divyamahajan@Divyas-laptop:~/code$ cd TECHNIQUES-TO-IMPLEMENT-WEB-CRAWLERS-USING-MULTI-THREADING/
.git/          Output/        SourceCode/    TestFiles/     setup_startup/
divyamahajan@Divyas-laptop:~/code$ cd TECHNIQUES-TO-IMPLEMENT-WEB-CRAWLERS-USING-MULTI-THREADING/SourceCode/
divyamahajan@Divyas-laptop:~/code/TECHNIQUES-TO-IMPLEMENT-WEB-CRAWLERS-USING-MULTI-THREADING/SourceCode$ ls
MultiThreadedCrawler.py  __pycache__  driver.py       monitorlock.py  semaphorelock.py  textprint.py
__init__.py              app          file_parser.py  scratchpad.py   startcrawler.py
divyamahajan@Divyas-laptop:~/code/TECHNIQUES-TO-IMPLEMENT-WEB-CRAWLERS-USING-MULTI-THREADING/SourceCode$ source ~/.serv-coder/bin/activate
(.serv-coder) divyamahajan@Divyas-laptop:~/code/TECHNIQUES-TO-IMPLEMENT-WEB-CRAWLERS-USING-MULTI-THREADING/SourceCode$ python3 driver.py
-----Welcome to Parallel Web Crawlers----
Please input maximum number (default:4) of Threads required to crawl the given URLs :

In this Web crawler mechanism you can use the different locking mechanisms
        1.Lock Free Crawlers
        2.Implementation of locks using Semaphore mutex
        3.Implementation of locks using Monitors

Would you like to automate the WebCrawler to run up to 1 to 4 threads for all available locks respectively(Y/N) :
Note:
        The crawler will be execute for once for 4 threads
        Please check Output/data_log_frontier10_2022-09-06.csv file for end record.

Please enter the type of lock (default: 1.Lockfree) you wish to implement from above options:

Please type 'Y' if you want to store the list of visited links:
```

Figure 12 Program running screenshot

B. *Pseudocodes of Algorithms*

1. *Optimistic*

Kung and Robinson proposed an algorithm for the optimistic concurrency control method [12]. They divided the execution of transaction into three phases:

i. Read Phase: data objects are read, the intended computation of the transaction is done, and writes are made on a temporary storage [13].
ii. Validation Phase: check to see if the writes made by the transaction violate the consistency of the database. The data stored in the temporary storage will be discarded, if the conducted check detects conflicts. Otherwise, the writing phase will write the data into the database [13].

The validation process can be described as [9]:

```
T: a transaction
ts: the highest sequence number at the start of T
tf: the highest sequence number at the beginning
of its validation phase

valid =true;
for t =ts+1 to tf do
  if (writeset(t) & readset[T]! = {}) then
      valid:=false;
if valid then
{
  write phase;
  increment counter,
  assign T a sequence number}
```

iii. Write Phase: If the validation phase passes successfully, writing will be performed to the database. If the validation phase fails to pass, all temporary written data will be aborted.[9]

2. Pessimistic

In the Pessimistic approach of concurrency control the lock is placed on the database's record for update other users/threads trying to access the same record can only access record as read-only or it would have to wait for a record to be 'unlocked' by the executing thread [13][14].

The pessimistic approach execution firstly executes the validate operation and if the validation is successful, the read, compute, and write operations are performed [17].

```
lock()
Begin Transaction()
if validation ='success'
     read()
     compute()
     write()
End
unlock()
```

3. Coarse-Grained

The Coarse-grained locking algorithm creates a frontier queue as a linked list, with each node having a URL value. The procedure to insert or remove a URL uses the same lock to make sure that only either of the method is performed. For insertion, the new node is appended to the head of the linked list and the new node is declared as the head. While for deletion, the head itself is removed and returned. The node at which this head was pointing, becomes the new head.

```
class UrlNode
{           value = url_link
next_node = null   //Data Type UrlNode object }
class FrontierLinkedList
{ head = null       // Data Type UrlNode object
procedure coarse_grained_insert (url):
{           lock.lock()
     new_url_node = UrlNode(value = url)
     if head = null
       {    head = new_url_node
         return  }
     current_url = head
     while current_url.next_node != null do
       { current_url = current_url.next_node }
      current_url.next_node = new_url_node
      lock.unlock()
 } }
procedure coarse_grained_get_url:
{           lock.lock()
    url = head.value
          head = head.next
          lock.unlock() } }
```

4. Fine-Grained

The Fine-grained locking algorithm creates a frontier queue as a linked list, with each node having a URL value. The procedure to insert or remove a URL uses multiple locks on different indices of the linked list to make sure trying to access a particular data does not block the other threads from accessing the different URL's. In the insertion method, the new node is appended to the head of the linked list and the new node is declared as the head. And for deletion, the head itself is removed and returned. This method ensures concurrency and avoids starvation of the threads.

```
class UrlNode
{ value = url_link
next_node = null //Data Type UrlNode object}
class FrontierLinkedList
{  head = null // Data Type UrlNode object
procedure fine_grained_insert (url,key):
{      new_url_node = UrlNode(value = url)
       if head == key
           {  lock.lock()
              head = new_url_node
              lock.unlock()
              return  }
       current_url = head
       while (current_url != null) do
           { if (current_url== key) //node found
                { lock.lock()
                   current_url = current_url.next_node
                   current_url.next_node = new_url_node
                   lock.unlock()
                   Break }
             current_url = current_url.next_node } }
procedure fine_grained_get_url:
{       current_url = head
        while (current_url != null) do
        {    if (current_url== key)
             {  lock.lock()
                 url = current_url.value
                 current_url = current_url.next_node
                 lock.unlock()
                  break }
              current_url = current_url.next_node } }
procedure fine_grained_delete(url,key):
{   if head == key
   {    lock.lock()
        current_url=head
        head = current_url.next_node
        current_url.remove
        lock.unlock ()
        break  }
   current_url
   while (current_url != null) do
   {   if (current_url== key) //node found
       {   Lock.lock()
           tmp_node = UrlNode(value = url)
           tmp_node=current_url.next_node

current_url.next_node=tmp_node.next_node
           tmp_node.remove
           Lock.unlock()
           break }
       current_url = current_url.next_node } } }
```

5. *Lock-Free*

Initially, in our project, the size of the frontier queue and the crawler depth is fixed to a constant. Fixing the crawler depth to a constant value helps to analyze the performance of the multi-threaded crawlers using a different number of threads. The input file with several URLs is validated. The valid URLs are pushed into the Seed URL list. The Seed URL list contains all the valid URLs. These URLs are pushed into the frontier queue, which is used to fetch the URLs to crawl the relevant web content. Since the crawling process is an expensive operation, the size of the frontier queue is constant and will prevent excessive seeding of the URLs into the frontier queue.

Further, the number of threads and the type of lock implementation required to crawl the given URLs is taken as input from the user. The URL from the frontier queue is retrieved and a check is made if it is already visited or not. If the URL is not visited and is the newly extracted URL from the crawler, it is inserted into the frontier queue for crawling. Further, the parsing of the links is done to get the absolute URL path of the given URL. Finally, the seed URL and the crawled pages are extracted from the above implementation.

```
//Initialization
FRONTIER_SIZE=30
MAX_RUNNING_TIME_SECONDS = 3
//Pseudocode
1. Take input of the seed URL from the user file
2. Validate URL
3. If the given URL is valid, push the URL into the Seed URL list, else exit
4. Input the number of threads required to crawl the given URL's
5. Input the type of lock the user would wish to implement (Lock-free, Semaphore, or Monitors)
6. Add the URL from the seed list to the frontier queue via a method add_url_to_frontier:
     • lock-free: frontier_queue.push ()
     • semaphore: semaphore.insertURL()
     • monitor: monitor.insertURL()
7. Fetch the URL from the frontier queue via a method get_url_from_frontier:
     • lock-free: frontier_queue.pop ()
     • semaphore: semaphore.removeURL()
     • monitor: monitor.removeURL()
8. Check whether this URL is already visited or not, if not, push this URL into the visited links list
9. Crawl this URL
10. Parse the raw data and extract meta data and links from the crawled URL
11. Check the new links with the visited links list, if unvisited, add them to the frontier queue via a method add_url_to_frontier
```

6. *Semaphore*

The second implementation explains the behavior of the semaphore locking in the web crawlers to ensure the mutual exclusion principle in a multi-threaded environment. The basic idea is to implement the two functions of the semaphore function P(), to decrement the value of the semaphore by 1, and function V (), to increment the value of the semaphore by 1. Initially, three semaphore variables named full_slots, empty_slots, and mutex are defined. To achieve the mutual exclusion on the shared resource i.e., the frontier queue, and to prevent the data from getting corrupted, during the insertion of the URL to the frontier queue, the p() function is used to decrease its positive counter or wait while it is not positive on the full slots and mutex variables. This would block the access of the other threads to access/modify the same URL. Once the URL is accessed, the v() function is used to increment counter. Similarly, during the deletion of the data from the frontier queue, make sure the URL to be deleted is not used by any other threads.

```
semaphore SemaphoreCrawler{
//the number of empty slots is decided by the fixed size of the frontier.
//an extra semaphore to implement mutual exclusion for the buffer.
semaphore empty_slots = FrontierSize,  full_slots=0, mutex=1
function insertURL(url){
//semaphore has a p() function to decrease its
      empty_slots.p( )
      mutex.p( )
      insert_url_to_frontier(url)
```

```
        //Semaphore has a v() function to increment
        the counter inside semaphore.
        mutex.v( )
        full_slots.v( )   }
  function removeURL{
        full_slots.p( )
        mutex.p( )
        url = remove_url_from_frontier
        mutex.v( )
        empty_slots.v( )   }}
  procedure add_url_to_frontier
     { while true do
          { url = crawled_parsed_filtered_url
            SemaphoreCrawler.insertURL(url) }}
  procedure get_url_from_frontier
     { while true do
          { url = SemaphoreCrawler.removeURL
            return url }}
```

7. Monitor
(TANENBAUM,2008)[7]

The above-mentioned algorithm explains the behavior of the multi-threaded web crawler system as Producer-Consumer using the monitors. Signal () and Wait () functions are used to achieve mutual exclusion. If the frontier queue is full, then the producer threads are made to wait until any slots are available to insert the URL. However, if the frontier queue is empty, then the producer threads are notified to insert the URLs into the frontier queue. Similarly, if the frontier queue is empty then consumer threads wait for the producer threads to produce.

```
monitor MonitorCrawler
{    condition full, empty
     integer count = 0
     function insert(url):
     {  acquire lock
          while count = FrontierSize
             {then wait(full)}
          insert_url_to_frontier(url)
          count = count + 1
           signal(empty)
           release lock}
     function remove:
     {  acquire lock
          while count = 0
             {then wait(empty)}
          url = remove_url_from_frontier
          count = count – 1
          signal(full)
          release lock} }
  procedure add_url_to_frontier
  {    while true do
             { url = crawled_parsed_filtered_url
               MonitorCrawler.insert(url) }}
  procedure get_url_from_frontier
  {    while true do
             { url = MonitorCrawler.remove
               return url}}
```

C. Instructions to run the program of this project

i. Add seed URLs into the TestURL.txt file in the TestFiles folder and compare the output of different locking techniques or number of threads with the same seeds.
ii. Initialize the Output folder before the test, otherwise, fresh data can be mixed with historic data .
iii. If each number of threads with the same locking technique is run n times for accurate result, the performance, p, is calculated by the mean of the results:
$$p = \frac{\sum_{i=1}^{n} num\ of\ links\ at\ time\ i}{n}$$
iv. The program must be run in an ideal environment. Because some websites are trying to detect web crawlers and block them. And API throttling also affects the efficiency of crawlers.

V. RESULTS

In this project, multithreaded web crawlers are implemented with lock-free, semaphore and monitor models separately. In each model, crawlers are working in a limited time, 3 seconds, and fetch URLs from a fixed size of 30 central frontier queue.

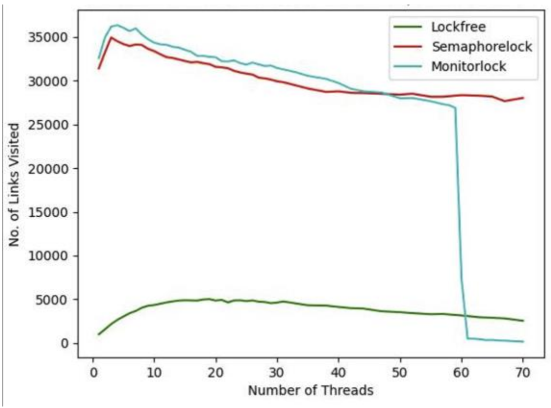


Figure 13 Performance with different locks

The table shows the number of links being crawled with different number of threads in those three models. It can be observed that several crawlers working together are more efficient than one crawler working alone. However, more threads do not always mean more efficiency.

Besides, crawlers do perform much better with locks than without locks.

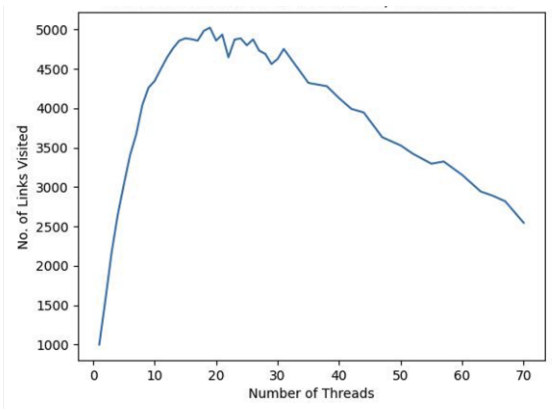


Figure 11 Performance of lock-free

The optimal number of links visited by lock-free architecture is only around 5000. If running crawlers with optimistic logic, the central frontier is overwritten frequently, and crawlers usually do repetitive crawling work when they grab the same URL from the unlock frontier. This greatly slows down the efficiency of crawling work. To correct it each time whenever collision happens is not feasible. This is the reason why lock-free architecture does not work as well as locking architectures do.

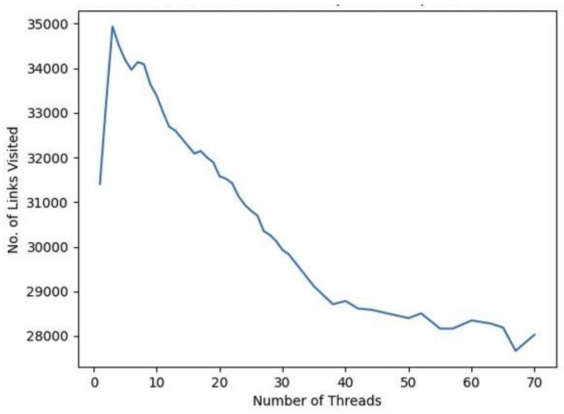


Figure 15 Performance of semaphore

Semaphore works very well and stable, but its performance still drops after the number of threads exceeds 8. This phenomenon can be explained by some reasons. First, the computer running this program has 8-performance-core processor, which means at most 8 threads can really run parallel. If there are more than 8 threads working, some of the threads only run concurrently. It means the operating system schedules them to run at different time slice. Under the circumstances, the improvement cannot be simply obtained from the increase of the number of threads. Then, the utilization of I/O block time, that is web page request time and response time, of each crawler becomes crucial. However, according to Amdahl's law, which states that "the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used" [11], there is a limit speedup time.

Adding further the second reason is when the number of threads is greater than the maximum number that can be ran in parallel, threads are scheduled to work concurrently with an order, and there is context switch time when each thread is uploaded and unloaded. The more the threads are, the more frequent context switch takes place. And the time slice allocated to each thread is expected to be shortened. To regularly interrupt a crawler while it is crawling but not I/O blocked is not a wise operation. This kind of synchronization overhead can slow down the overall performance.

Although semaphore performs well, it is a very low-level primitive. Small error brings whole system to grinding halt while it is difficult to debug. Monitor is much easier to be programmed and avoid unexpected errors.[7]

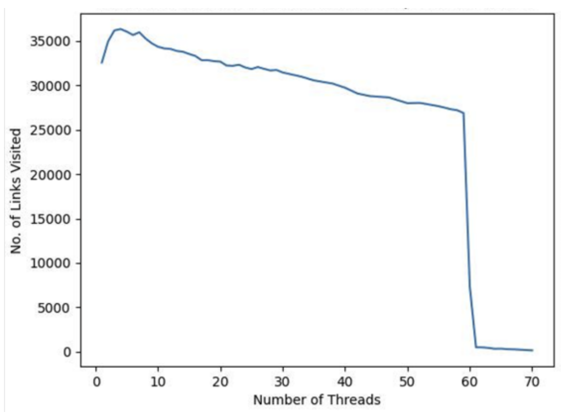


Figure 16 Performance of monitor

The monitor model works similarly to semaphore model at first. However, the performance drops significantly when the number of threads reaches twice the frontier size. This phenomenon can also be observed when the frontier size changes.

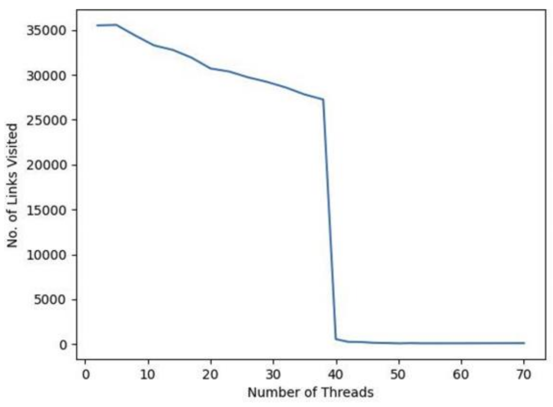


Figure 17 Performance of monitor with frontier size 20

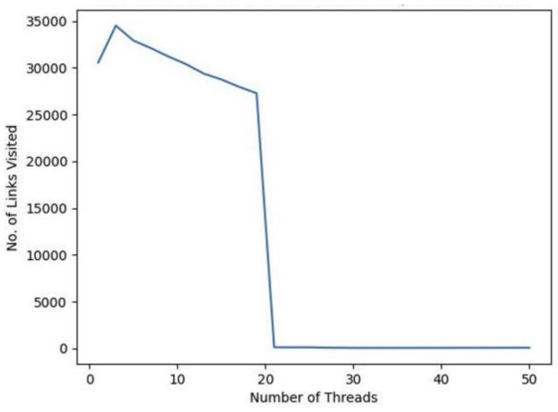


Figure 18 Performance of monitor with frontier size 10

This interesting observation needs to be explored in future scope. The performance of monitor before the plunge also shows the advantages of locking technique and multithreaded web crawlers.

VI. CONCLUSION

This paper proposes various techniques to implement the multi-threaded web crawlers. It aims to analyze the performance of crawlers with architectures of lock-free, semaphore and monitor. Experimental study ascertains the importance of using the multithreaded web crawler system and usage of the locking techniques for high performance. Multithreaded web crawlers are working in a controlled environment. It is like a classical producer-consumer problem.

When the number of threads is constant, the program shows that the semaphore locking works better than the other two, not only efficient, but also stable. But in the real world, semaphore is not widely used. When the problem becomes complicated, semaphores should be carefully arranged in case of deadlock, starvation, or any other unexpected problems.[7] From this point of view, monitor is much easier to apply. It is a higher-level synchronization primitive.[7] However, performance will decrease significantly while too many threads are working concurrently, regardless of what architecture is used. This is mainly caused by the limit of maximum number of parallel threads and synchronization overhead.

VII. FUTURE SCOPE

This research can be further optimized by using the fine-grained locking in Multi-threaded Web Crawlers. The usage of the lock granularity can cause lock contention issues. However, compared to coarse-grained locking mechanism, fine-grained locking mechanism can mitigate lock contention issues but difficult to use. Also, this experiment can be implemented using the linked list data structure for the frontier queue as the linked list which is a dynamic data structure and can grow and shrink at runtime by allocating and deallocating memory. So, there would be no need to initialize the size for the frontier queue in prior.

Further, in the experiments conducted, we could observe a significant drop in the performance of the monitor model when the number of threads reaches twice the frontier size. This behavior can be observed when the frontier size changes. However, this area needs to be explored in the future scope.

VIII. REFERENCES

[1] Dhar, T., Mazumder, S., Dhar, S., Karak, S., & Chatterjee, D. (2021, October). An Approach to Design and Implement Parallel Web Crawler. In 2021 International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON) (pp. 1-4). IEEE.

[2] Perisetla, K. K. (2012). Mutual exclusion principle for multithreaded web crawlers. Edit Preface, 3(9).

[3] Sultanov, A., Protsyk, M., Kuzyshyn, M., Omelkina, D., Shevchuk, V., & Farenyuk, O. (2021, September). Comparison of performance of the popular approaches to implementing parallel crawlers. In 2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT) (Vol. 1, pp. 349-352). IEEE.

[4] Liu, H., Hu, T., & Qiu, Z. (2017, September). Automatic fine-grained locking generation for shared data structures. In 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE) (pp. 1-8). IEEE.

[5] Meegahapola, L., Alwis, R., Heshan, E., Mallawaarachchi, V., Meedeniya, D., & Jayarathna, S. (2017, August). Adaptive technique for web page change detection using multi-threaded crawlers. In 2017 Seventh International Conference on Innovative Computing Technology (INTECH) (pp. 120-125). IEEE.

[6] Wang, H., Wang, Z., Sun, J., Liu, S., Sadiq, A., & Li, Y. F. (2020, September). Towards generating thread-safe classes automatically. In 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 943-955). IEEE.

[7] Tanenbaum, A. (2009). Modern operating systems. Pearson Education, Inc.,

[8] Multithreaded crawler in Python. GeeksforGeeks. (2021, September 23). Retrieved May 12, 2022, from https://www.geeksforgeeks.org/multithreaded-crawler-in-python/

[9] Optimistic algorithm. (n.d.). Retrieved May 18, 2022, from https://courses.cs.vt.edu/~cs5204/fall99/distributedSys/optimistic.html#:~:text=Optimistic%20Concurrency%20Control%20Algorithms%20are,that%20no%20conflicts%20have%20occured.

[10] Herlihy, M., & Shavit, N. (2008). The Art of Multiprocessor Programming [SMP].

[11] Hill, M. D., & Marty, M. R. (2008). Amdahl's law in the multicore era. Computer, 41(7), 33-38.

[12] “Optimistic Concurrency Control.” Wikipedia, Wikimedia Foundation, 19 Apr. 2022, https://en.wikipedia.org/wiki/Optimistic_concurrency_control.

[13] 11, Emnuel May, et al. “Optimistic vs. Pessimistic Locking.” Vlad Mihalcea, 16 Feb. 2022, https://vladmihalcea.com/optimistic-vs-pessimistic-locking/.

[14] “Record Locking.” Wikipedia, Wikimedia Foundation, 23 Oct. 2021, https://en.wikipedia.org/wiki/Record_locking.

[15] Coarse-Grained and Fine-Grained Locking - Lunds Tekniska Högskola. https://fileadmin.cs.lth.se/cs/Education/EDA015F/2013/Herlihy4-5-presentation.pdf.

[16] Coarse-Grained, Fine-Grained, and Lock-Free ... - UNLV Libraries. https://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=4733&context=thesesdissertations.

[17] “Difference between Pessimistic Approach and Optimistic Approach in DBMS.” GeeksforGeeks, 29 June 2021, https://www.geeksforgeeks.org/difference-between-pessimistic-approach-and-optimistic-approach-in-dbms/.

IX. RETROSPECTIVE

The research paper [1] provides details on some of the crucial elements of a multi-threaded web crawler and some of the algorithmic particulars. The paper describes the parallelization of the crawler and how the crawler can be used to download the web pages. Another research paper [2] explains how mutual exclusion principle be achieved in a multi-threaded web crawler system and how each crawler thread in a multi-threaded system can be used to access the frontier queue in a synchronized manner to avoid the potential deadlock situation and achieve higher performance. Research paper [4] explains the fine-grained locking mechanism and how it can automate to achieve synchronization. Further [5] provides details on the methods to detect the change frequency of web pages efficiently in a Multi-threaded Web Crawler system. The experiments conducted in this paper highlight the importance of the usage of the multi-threaded crawler system and optimization of the change detection process which can be used in a high-performance server.

However, we tried to implement the lock techniques during the insertion of the URL and fetching of the URL from the frontier queue. Our experiment on the “Techniques to implement the web crawlers using multi-threading" explores the various locking mechanisms such as lock-free, semaphore mutex and monitors and its implementation in the multi-threaded web crawler system. The data-points collected from the various test cases can be used to analyze and compare the thread performance in a multi-threaded environment. This project experiment provides:

- Study on various research papers on multi-threaded applications to achieve mutual exclusions provided in-depth understanding of the multi-threads and their implementation in the web crawler system.
- It helps to understand various locking mechanisms such as optimistic locking, pessimistic locking, coarse-grained, and fine-grained locking mechanisms. This study helped to design our WebCrawler system to perform an analysis of the factors affecting the performance of the system.
- It also displayed individual commitment to a group effort by inculcating individuals' ideas to design and develop a product to achieve our project goals.
- This experiment provided an opportunity to explore and understand the working and design of the web crawler system.
- Also, it helped to understand and explore the various libraries in the python programming language to implement the multi-threading web crawler system using different locking mechanisms like the producer consumer environment.

As with any project there is always room for improvement and given more resources in terms of time, this is what we could have done differently:

- Try to understand and simulate the project in a multi-processor environment.
- Implement the fine-grained locking in the multi-threaded environment.
- Try implementing the multi-threaded web crawler system using the Thread class i.e as a single task instead of Thread pools as this would dispatch many similar tasks.
- Try implementing dynamic memory management data structures such as linked lists.
- Try to implement and automate the process testing the system using various data points collected and calculate the rate of successfully crawled pages to return the top results.
- Try to explore the reason behind the significant drop in the monitor performance when the number of threads reaches twice the frontier size.