Adaptive Technique for Web Page Change Detection using Multi-threaded Crawlers

Lakmal Meegahapola, Roshan Alwis,
Eranga Heshan, Vijini Mallawaarachchi,
Dulani Meedeniya
Department of Computer Science & Engineering,
University of Moratuwa, Sri Lanka
{lakmalbuddikalucky.13, alwisroshan.13, eranga.13, vijini.13,
dulanim}@cse.mrt.ac.lk

Sampath Jayarathna
Department of Computer Science,
California State Polytechnic University
Pomona, CA 91768
ukjayarathna@cpp.edu

***Abstract*— World Wide Web is getting dense as many new web pages and resources are created on a daily basis. Keeping track of the changes in the web content has become an immense challenge and is a research problem with a great significance. Even the search engines require to detect changes in the web to keep search indexes up to date. Numerous researches have been carried out on optimizing the change detection algorithms. This paper presents a methodology named Multi-Threaded Crawler for Change Detection of Web (MTCCDW), which is inspired from the producer-consumer problem. The suggested change detection process mainly analyses the performances and suggests a tread-based implementation process for the optimisation of the changed detection process. The experimental results show that the proposed methodology is capable of reducing the effective time to detect changes in a web page by 93.51%.**

Keywords*— *web page change detection, web crawling, multi-threading; producer-consumer problem; change detection;

I. INTRODUCTION

The World Wide Web (WWW) keeps on expanding frequently as many tools and services become available to create and maintain web content. Various websites, including news websites and different themed blogs are created in order to share knowledge and educate the general public about current affairs. Keeping track of the changes occurring in this web content has shown many challenges, as there are numerous web pages keep on changing frequently. Change Detection and Notification (CDN) systems have made the process of keeping track of changes to web pages more efficient and less tedious [1]. Furthermore, navigating through the different web pages and finding the required content has become a difficult and time consuming task. Search engines [2] [3] have made the task of searching for web content easier and less tedious as they search the WWW for relevant [4] web pages and content. The major component of all of these systems is the web crawler, which goes through all the web pages and downloads them to detect whether changes have occurred. This information will be used for the indexing process [5]. A web crawler mainly identifies whether the changes have occurred in the crawled web pages. Further, with the large datasets, the need for analysing the performance and power consumption arises [6].

In many existing change detection mechanisms, all the tasks including retrieval of current versions of web pages and comparing with saved versions to detect changes are carried out as a single process. This can affect the efficiency and performance of the process. This paper addresses this issue from an architectural perspective. We divide the change detection process in to sub-tasks and assign them to different threads, rather than having a single bulk process, which will result in a multi-threaded solution. Since there are many threads reading and writing to a common location, the well-known producer-consumer problem [7] can be incorporated to the process of change detection.

Many parallel threads can be used to rapidly solve large problems [8]. In this paper, we model the crawler thread as the producer, which fetches the current version of the web page and saves them in a queue. The thread with the change detection module can be modelled as the consumer, which consumes the fetched web pages in order to detect changes by comparing with the existing version. Even though we keep on increasing the number of threads, the process will reach a certain limit where no significant improvement will be shown. Hence, it is necessary to ensure that the optimum number of threads will run in order to produce an efficient change detection process. However, it is a challenge to schedule tasks for threads during parallelization [9].

This paper presents a methodology named Multi-Threaded Crawler for Change Detection of Web (MTCCDW) to detect the change frequency of web pages efficiently. Section II outlines the related work in the field of consideration. Section III describes the design and methodology. Section IV explains the experimental study which we have carried out and Section V presents the obtained experimental results. Finally, Section VI concludes the paper with the inferences obtained from the results and emphasizes on the importance of the research.

II. BACKGROUND

Many studies have been carried out to improve the efficiency of the change detection process by proposing many improved algorithms and techniques. Nadaraj [10], has described an approach for distributed content aggregation and change detection for web content using client resources. In this

978-1-5090-3989-0/17/$31.00 ©2017 IEEE

approach, the consumers obtain the data from a queue and run the web crawlers on the working machine. It distributes the work among consumers and the results will be aggregated, improving the efficiency of the crawling mechanism. It also reduces the coupling of web crawlers to a particular machine, allowing them to operate in a distributed network. It can be considered as a scalable content classification approach. Bloom filters have been used to find the duplicate URLs and content in the site. However, bloom filters only confirm that the URL was not visited before. Furthermore, still the process of change detection can be delayed as the bottleneck becomes the retrieval of the current versions by crawling. Another interesting work on detecting changes in distributed and collaborative web data collections and notification method is presented in [11]. This is mainly used by search engines to determine the schedule to crawl the web pages and build the indexes. PageRank values using *shash* tool is considered to detect the near duplicates. This provides fast change detection with a low maintenance cost. However, if there are many requests at a given time, due to heavy usage, the system may not be able to process those in real time.

Kausar et al. [12], propose a system based on parallel web crawling using mobile agents. The web crawler is considered to be mobile as it can migrate to the data source before starting the crawling process. As indicated, the main advantage of a parallel web crawler based on mobile agents is that it reduces network load and traffic as the analysis part of the crawling process is done locally. However, issues such as running out of space in a server due to overloading of crawlers can arise and this may affect the synchronization of the crawling process. An architecture for a parallel crawling of the web pages using multiple machines and integrating the trivial issues of crawling is presented in [13]. The authors have provided a three-step algorithm for detecting web page changes. The server has a unique method for distribution of URLs to clients after determination of their priority index. However, the clients are server nodes themselves. Hence the number of server nodes has to increase when scaling the system that results in high cost.

Shkapenyuk and Suel [14], describe the design and implementation of a high-performance distributed web crawler, which runs on workstations. It can be adapted to various crawling applications. This system is partitioned in to two main components; crawling system and crawling application. The crawling system consists of a *crawl manager*, one or more *downloaders*, and one or more *DNS resolvers*. The change detection process is being carried out by the *crawling application*. It parses each downloaded page for hyperlinks and checks whether these URLs have already been encountered before. However, the system parses for hyperlinks and not for indexed terms and this can be conflicting when detecting content changes of web pages. Furthermore, the authors have highlighted the need for highly efficient crawling systems.

Another distributed web crawler model based on cloud computing and parallel implementation has been proposed in [15]. The model is based on the master-slave architecture and consists of several components including a cloud storage system, scheduling system, distributed processing system and multiple crawling systems. The web pages fetched by the crawling systems are processed in the distributed processing system. The authors have implemented the crawling system to crawl in parallel, which in term reduces the network usage and CPU resources. However, there exists the limitation of load balancing between different nodes during the crawling process.

Mali and Meshram [16], propose a focused web crawler with a page change detection policy. They have focused on the page selection policy and page revisit policy. The crawler first explores the relevance of the page and checks whether the structure or the content has changed. Then, the crawler updates the URL repository. Structural changes are detected by using changes for the HTML tags, while text changes are identified by converting the text to a particular code and comparing it with a previously saved version. However, still the process of change detection can be delayed as the bottleneck becomes the retrieval of the web pages by the crawling process.

Furthermore, Sebastian [17] proposes another focused web crawler for information retrieval. The specialty of this crawler is that it uses genetic algorithms. This web crawler focusses on obtaining information regarding a specific subject. As it retrieves only the required data and learn from the past traversal experiences, it can save sufficient time and resources as irrelevant data are not processed. An advantage of this adaptive mechanism is that the process can be left unsupervised as the crawlers have the ability to learn. The author has mentioned the importance of having a central control mechanism to coordinate the crawlers running in parallel and to pre-vent redundancy. However, if this crawler was used in commercial web search engines, issues may arise as there are web pages belonging to a vast number of subjects.

As discussed above, a wide range of research has been carried out to enhance the web page change detection by addressing many aspects of the process such as the crawler scheduling, change detection algorithms and the architecture itself. Furthermore, most of the researches regarding web crawler efficiency have been focused around parallelizing the crawling process (parallel web crawlers). However, only a limited amount of research has been carried out regarding the use of a multi-threaded approach to be incorporated in the web page change detection process.

III. System Design and Methodology

Fig. 1, shows a high level view of a server in a general change detection system. This process is sequential with respect to the web pages in the browsing list and a schedule is prepared to crawl and detect changes. From the schedule, each web page is crawled, fetched and compared with its old version using the *Change Detector*. The process consists of three main steps as shown in Fig. 2.

1. Crawling - Retrieving the current version of the web page from WWW.
2. Fetching - Retrieving the old version of the web page from Version Repository where old versions of the web page are stored.
3. Change Detection - Comparing the two versions using a suitable comparison mechanism.

In step 1, the system retrieves the web page in which changes are required to be tracked. This requires the internet connection of the system and depends on the network usage of the machine. The time taken to complete this step is higher compared to the time taken for other steps as shown in Fig. 2. In step 2, the previous version of the fetched web page is retrieved from the database and consumes less time compared to step 1. The speed of this process mainly depends on the type of the database and whether the server is remote or local. As the change detection process is running locally on a server, the database would be local to the program. Hence, the time taken in this step will be less. In step 3, the two versions are checked for changes.

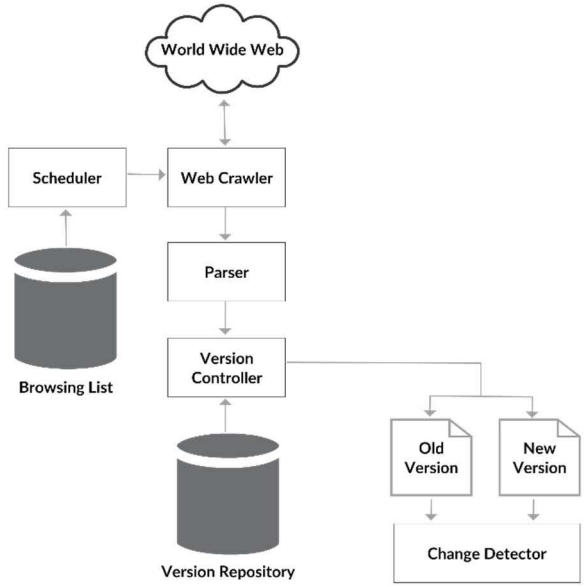

Fig. 1. High level view of an existing change detection system.

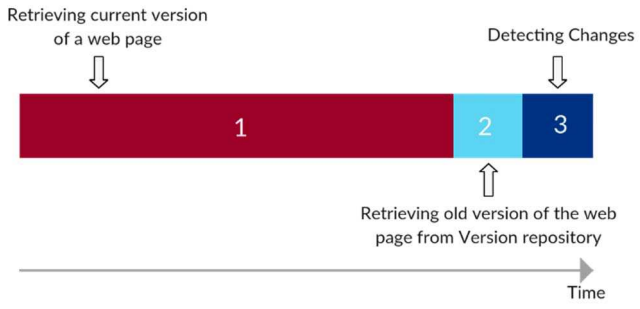

Fig. 2. Three main parts of detecting changes of a web page

Most of the current research is focused on this step and various implementations and algorithms are available to increase the efficiency and accuracy of this step. Although step 3 is optimized in the best possible manner, still the entire process of change detection mainly depends on the time taken by step 1. Hence it is clear that to keep the entire process of change detection of a web page at a certain pace, step 1 has to be completed in a high pace compared to step 2 and step 3.

The solution to the problem discussed above, lies in the MTCCDW methodology, which divides the change detection process into three separate sections. Each section carries out the task performed in a step in the normal change detection procedure. However, the difference between the existing change detection mechanism and the MTCCDW methodology is that each of the sections uses a number of threads to perform the task as shown in Fig. 3. Following terms are used here.

1. Section 1 – carries out the process of retrieving current versions of the web pages.
2. Section 2 – carries out the process of retrieving old versions from the version repository.
3. Section 3 – compares the two versions and detects the changes.

Consider a scenario where, section 1 uses N number of threads, section 2 uses M number of threads and section 3 uses K number of threads. N, M and K are non-zero positive integers. These 3 numbers can be chosen in an optimum manner to make the entire process of detecting changes in web pages more efficient.

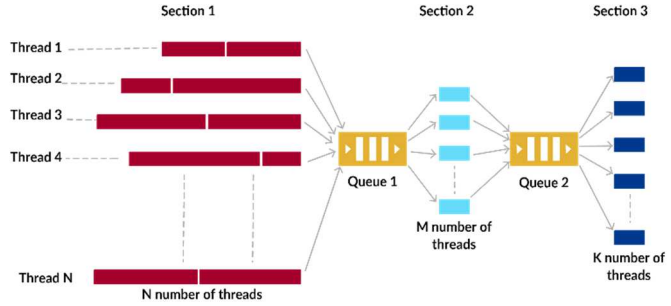

Fig. 3. Multi-threading usage for optimising change detection in MTCCDW.

As shown in Fig. 3, three sections and two queues were used in the initial implementation. Section 1 fetches the web page, creates an object together with web page ID and puts it in the thread-safe queue, Queue 1. Section 2 runs M number of threads and each of these threads fetch objects from the queue, fetch the required previous version from the version repository, create a new object with the two versions of the web page and push it into the thread-safe queue, Queue 2. Sections 3 which is implemented with K threads, fetches objects from the Queue 2 and does the comparison to detect changes. With sufficiently large N, this system ensures that the Virtual Machine in which the server runs is working optimally without being idle.

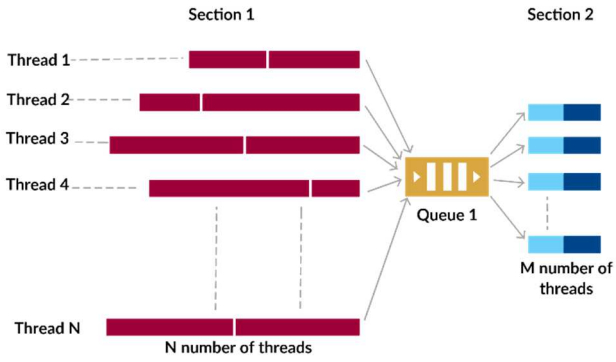

Fig. 4. Multi-threading implementation for an optimised MTCCDW.

Considering the obtained performance measures (explained in Section IV), this structure can be further optimized as shown

in Fig. 4. It was observed that the optimized version is obtained in a scenario where M is equal in value to K. In the optimized version, section 2 and section 3 of the previous version is combined to create section 2. Here there is only one thread-safe queue and section 2 handles both fetching web pages from the version repository and detecting changes. This version of MTCCDW removes the overhead of objects moving between two queues and hence the whole process is optimized.

IV. Experimental Study

A. Implementation

Initially, the experiment was performed with a single thread and obtained the results using the average of 5 iterations to complete detecting changes of 6540 web pages. Then we used the thread configurations shown in Fig. 3, and Fig 4, using variables N, M and K. From the results of the configuration in Fig. 3, the best results were obtained when M=K, as shown in Table II. Hence, section 2 and section 3 of Fig. 3, were combined for the 3rd configuration as in Fig. 4.

This change detection process is based on the producer-consumer problem [7], which is a well-known example of a multi-process synchronization problem. This consists of two processes, the producer and the consumer, which share a common, fixed-sized buffer. The producer produces data and feeds to the buffer and continues the production process. Simultaneously, the consumer consumes the data from the buffer. In such a scenario, it is a challenge to ensure that the producer will not add data to the buffer if it is full and the consumer will not try to consume data when the buffer is empty. The change detection process modelled this scenario.

While changing the number of producer threads (N) against the number of consumer threads (M), we tried to find the optimum N and M values which will result the lowest completion time of detecting changes of 6540 different webpages. Since detecting changes (consumer's job) takes less time than crawling (producer's job), two more conditions to N and M such that M <= N/4 and M <= 15.

These conditions were included to ensure that the consumers would not starve. The value of N was limited to N<=100, by considering the network usage for the process of crawling and available data rate in the virtual machine. For all the N and M values, the experiment is continued for 5 iterations. We used 5 iterations to be within a confidence level of 95% with a standard deviation of 3.21. Hence, following three configurations were simulated during the experiment study.

1. Configuration 1: Single threaded crawler
2. Configuration 2: Multi-threaded crawler with different values for variables N, M and K. (As in Fig. 3.)
3. Configuration 3: Multi-threaded crawler with different values for variables N and M. (As in Fig. 4.)

Here, we tested how currently available implementations of change detection systems work against MTCCDW in the initial version as in Fig. 3, and the optimized version as in Fig. 4, using 6540 different web pages. Effective time to carry out the process of change detection of a web page was measured in each configuration and shown in Eq. (1).

$$\text{Effective Time} = \frac{\text{Time difference of start and end of change detection}}{\text{Number of Web Pages}} \quad (1)$$

The experiments were carried out on a virtual server hosted in Azure private cloud. It had Linux Ubuntu (kernel 16.04 amd64) and was running on 64-bit IntelTM Intel Xeon E312xx (Sandy Bridge) which operates at 2.70GHz. It had 4 CPU sockets, with 2 cores each. L1(d/i) and L2 caches were 512KB 0200and 4MB respectively. Its primary memory (RAM) was 16GB and secondary memory (HDD) was 100GB.

B. Pseudo Codes of Algorithms in MTCCDW

The pseudocode given in Algorithm 1 explains the MTCCDW process. This indicates the basic initialization required for both producer and consumer algorithms. Following are the terminology used for the initialization.

- linkQueue: Contains URLs which were not crawled
- documentQueue: Contains crawled documents
- producerLock: A lock to manage producer threads
- consumerLock: A lock to manage consumer threads
- capacity: Maximum capacity of the document queue (producers will not exceed the production more than this limit. When producer reach this limit, it waits until consumers to consume.)
- producerCount: Number of concurrent producers

Algorithm 1 Initialization

```
1. linkQueue = Queue(String)
2. documentQueue = Queue(Document)
3. producerLock = Lock()
4. consumerLock = Lock()
5. capacity = 500
6. producerCount = 100
```

The *linkQueue* is a queue of strings that contains URLs. It is only used among producers for fetching URLs to crawl the relevant web content. The *documenQueue* is shared among both producers and consumers. Producers will store crawled documents in the *documentQueue,* while consumers will consume the produced documents from the same queue. Since the crawling process is an expensive operation, this queue has a limited size that will prevent producers from excess production. The capacity parameter decides this maximum size of the *documentQueue*. The *producerLock* will manage the concurrent access of producers and *consumerLock* will handle the concurrent access of consumers.

Algorithm 2 explains the work carried out by one producer thread in parallel crawling. Here, the term *link* refers a URL to crawl and *document* is the returning document by the crawler. Producer function takes a link from the *linkQueue* and feeds that link to the crawling process. The crawled document will be stored in the *documentQueue*. The *producerLock* is used to handle the concurrent access to the *linkQueue,* which is a

shared resource among producer threads. Moreover, *producerLock* works with *consumerLock* when updating the *documentQueue,* which is shared between producers and consumers. If the *linkQueue* is empty, producers will wait until it fills. If the *documentQueue* is filled, the producers will wait untill consumers to consume. Since the crawling part is the slowest process, we have not synchronized that process which can block other producer threads and consumers. Finally, when *documentQueue* is updated, it will notify consumer threads, which can be waiting on the empty *documentQueue*.

Algorithm 2 Producer Algorithm

```
Require: URLs to crawl
Ensure:
1. Producer():
2.   while(True):
3.     link = NIL
4.     document = NIL
5.     producerLock.acquire()
6.     if linkQueue is empty:
7.         producerLock.acquire()
8.     if documentQueue.size() < capacity - producerCount + 1:
9.         link = documentQueue.dequeue()
10.    else:
11.        producerLock.acquire()
12.    producerLock.release()
13.    if link is not NIL:
14.        document = crawl(link)
15.    producerLock.acquire()
16.    if document is not NIL:
17.        productQueue.enqueue(document)
18.        consumerLock.release()
19.    producerLock.release()
```

Algorithm 3 explains the behaviour of a consumer thread in a multi-threaded environment. Consumer threads are working on detecting changes for the documents available in the *documentQueue*. If the *documentQueue* is empty, then consumer threads wait for the producer threads to produce. If a consumer consumes a document, it notifies the producer threads which is waiting on the filled document queue.

Algorithm 3 Consumer Algorithm

```
Require: Documents to detect changes
Ensure:
1.  Consumer():
2.      document;
3.      while(True):
4.          document = NIL
5.          consumerLock.acquire()
6.          while documentQueue is empty:
7.              consumerLock.acquire()
8.          producerLock.acquire()
9.          document = documentQueue.dequeue()
10.         producerLock.release()
11.         if document is not NIL:
12.             process(document)
```

V. Results and Discussion

Here, we have considered the minimum effective time and the optimum effective time. In minimum effective time, the effective times will be calculated for each N by varying M and K. Minimum effective time would be the minimum time out of the set of times we get by varying M and K for a particular N. Optimum effective time is the minimum out of the set of minimum effective times, where there is a minimum effective time for each N. As shown in Table 1, the effective time for crawling a web page using a single thread was 911.05 ms in configuration 1. It is extremely high compared to the values obtained using multi-threaded crawling in other configurations.

TABLE I. Optimum Effective Time for Change Detection in Each of the 3 Configurations

	Configuration 1	Configuration 2	Configuration 3
Effective time for detecting changes of a web page in the optimum scenario	911.05ms	60.05ms (N = 78, M = 6, K = 6)	59.19ms (N = 78, M = 11)

We obtained the minimum average/ effective time for change detection by varying M and K values, for each N from 1 to 100. These minimum effective times were plotted against N as shown in Fig. 5. For each of the minimum times, we observed how M varies with K. As shown in Table 2, M and K had equal values for 73% of the times. The difference between M and K was just 2 threads or less in 16% of the instances.

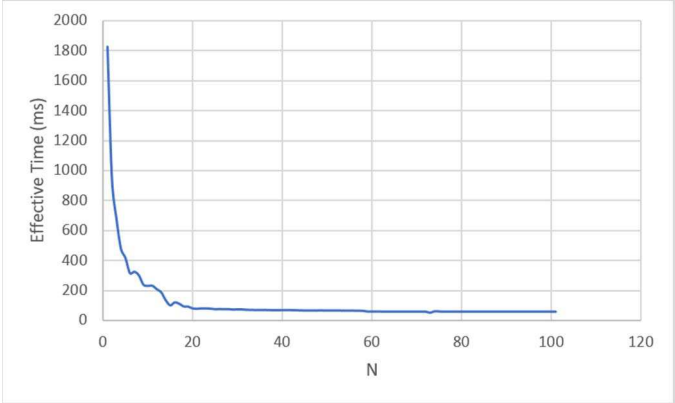

Fig. 5. Variation of effective time with N in the configuration 2 (MTCCDW without optimization)

TABLE II. Variation of M & K in Instances Where Minimum Effective Time Was Obtained for Each N

	M<<K	M<=K+2	M=K	M>=K-2	K>>M
% of cases	4%	5%	73%	11%	7%

According to the results shown in Table II, the minimum effective times were obtained when M is closer to K. Hence, the number of threads used in section 2 and section 3 of the configuration 2 given in Fig. 3. can be merged because the number of threads M and K are equal. Configuration 3 was obtained with this merging. In configuration 3, the minimum effective time out of the values for various M was plotted for each N, as shown in Fig. 6. From the effective times obtained in configuration 3, a clear optimization of effective times could be seen for each N. Let's consider the following terminology in regard to both configuration 2 and configuration 3. We

observed the relationship between minimum effective times for all N in both configurations 2 and 3.

T_{config_2} – Effective time in configuration 2 for a given N

T_{config_3} – Effective time in configuration 3 for a given N

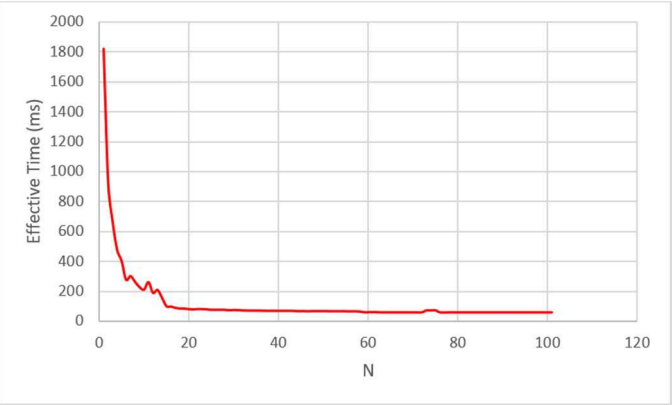

Fig. 6. Variation of Effective time with N in the configuration 3 (MTCCDW with optimization)

TABLE III. VARIATION OF M IN INSTANCES WHERE MINIMUM EFFECTIVE TIME WAS OBTAINED FOR EACH N

	$T_{config\_2} < T_{config\_3}$	$T_{config\_2} = T_{config\_3}$	$T_{config\_2} > T_{config\_3}$
% of cases	4%	0%	96%

In 96% of the cases, the time for change detection with each N was less in configuration 3 than in configuration 2. Further from configuration 2 to 3, an effective time reduction of 1.43% was observed as shown in Table 1 considering the optimum cases. If the effective time differences for all the N values are considered for both configurations, still an effective time reduction 0.1727% could be obtained. Compared to configuration 1, the effective time in configuration 3 has been reduced by 93.51%. From all these facts it could be concluded that MTCCDW with 2 sections, is an efficient way in which multi-threaded crawling can be used for change detection.

VI. CONCLUSION

This paper proposes a multi-threaded crawler for change detection of web to optimize change detection of web pages using multi-threading. An efficient change detection system could be useful in creating programs which use existing high performance servers in an optimum manner. The experimental study ascertains the importance of using multi-threading for the optimisation of the change detection process. The proposed methodology separates the entire process into two main sections, web crawling and change detection; where multi-threading is used in both processes. Further, the results show that when the number of threads for web crawling increases, the effective time for change detection gets reduced and saturates at a certain level after a particular number of threads. This could be mainly due to the network usage limitations of the experimental environment. Another limiting factor which affects the change detection process is the thread-safe queue, which is used in between web crawling and detecting changes. This research can be extended to have multiple thread-safe queues replacing the single thread-safe queue, which is used currently and observe how performance can be improved without allowing any threads to starve.

REFERENCES

[1] S. Chakravarthy and S. Hara, "Automating Change Detection and Notification of Web Pages (Invited Paper)," in 17th International Workshop on Database and Expert Systems Applications, Krakow, Poland, 2006, pp. 465-469.

[2] A. Cho *et al.*, "Searching the Web," ACM Transactions on Internet Technology, vol. 1, no. 1, pp. 2-43, 2001.

[3] L. Hung and C. Y. Lin, "Efficient parallelised search engine based on virtual cluster," International Journal of Computational Science and Engineering, vol. 12, no. 1, pp. 53-57, 2016.

[4] M. Kheira *et al.*, "Collection and Selection Based Relevant Degrees Of Documents," Journal of Digital Information Management, vol. 13, no. 2, pp. 110-119, 2015.

[5] V. M. Prieto, M. Álvarez and F. Cacheda, "Soft-404 Pages, A Crawling Problem," Journal of Digital Information Management, vol. 12, no. 2, pp. 73-92, 2014.

[6] Y. Khaliq *et al.*, "Calculation of CPU performance, power and cost using Hadoop," in 6th International Conference on Innovative Computing Technology, 2016, pp. 122-127.

[7] A. Hamroush and H. Tawfik, "Synchronized Information in the Producer-Consumer Problem," Journal of Socioeconomic Engineering, no. 2, pp. 25-30, 2014.

[8] C. Wu *et al.*, "Adjusting Thread Parallelism Dynamically to Accelerate Dynamic Programming with Irregular Workload Distribution on GPGPUs," Journal of Grid and High Performance Computing, vol. 6, no. 1, pp. 1-20, 2014.

[9] S. K. Singh and D. P. Vidyarthi, "Independent Tasks Scheduling using Parallel PSO in Multiprocessor Systems," Journal of Grid and High Performance Computing, vol. 7, no. 2, pp. 1-17, 2015.

[10] S. Nadaraj, "Distributed Content Aggregation& Content Change Detection using Bloom Filters," Journal of Computer Science and Information Technologies, vol. 7, no. 2, pp. 745-748, 2016.

[11] V. M. Prieto *et al.*, "Distributed and Collaborative Web Change Detection System," Computer Science and Information Systems, vol. 12, no. 1, pp. 91-114, 2015.

[12] M. Kausar *et al.*, "An Effective Parallel Web Crawler based on Mobile Agent and Incremental Crawling," Journal of Industrial and Intelligent Information, vol. 1, no. 2, pp. 86-90, 2013.

[13] D. Yadav *et al.*, "Architecture for Parallel Crawling and Algorithm for Change Detection in Web Pages," in 10th International Conference on Information Technology, 2007, pp. 258-264.

[14] V. Shkapenyuk and T. Suel, "Design and Implementation of a High-Performance Distributed Web Crawler," in 18th International Conference on Data Engineering (ICDE), San Jose, CA, USA, 2002, pp. 357-368.

[15] J. Yu *et al.*, "A Distributed Web Crawler Model based on Cloud Computing," in 2nd Information Technology and Mechatronics Engineering Conference, 2016, pp. 276-279.

[16] S. Mali and B. Meshram, "Focused Web Crawler with Page Change Detection Policy," in 2nd International Conference and workshop on Emerging Trends in Technology, 2011, pp. 51-57.

[17] K. Sebastian, "A Framework for adaptive focused web crawling and information retrieval using genetic algorithms," Journal for Computer Technology & Applications, vol. 6, no. 5, pp. 770-778, 2015.