# Multi-Threaded Web Crawler And Optimized Indexing

**Group members**

Deepshika Chand (2210110250)

Chahit Luthra (2210110236)

Aakrisht Narang (2210110097)

## Literature Review / State-of-the-Art

### 1. Experimental Performance Analysis of Web Crawlers Using Single and Multi-Threaded Web Crawling and Indexing Algorithm (A.K. Sharma, 2021)

This research focuses on evaluating the performance of **multi-threaded web crawlers** and their **efficiency in indexing crawled pages**. The authors compare **single-threaded and multi-threaded crawling models**, using metrics such as execution time, harvest ratio, precision, and recall. The study finds that **multi-threaded crawlers significantly reduce execution time**, improving data retrieval efficiency. Furthermore, it introduces a **hierarchical clustering-based indexing approach**, making it directly applicable to web search engines.

**Link to the research paper -**
https://www.researchgate.net/publication/343356518_Experimental_performance_analysis_of_web_crawlers_using_single_and_Multi-Threaded_web_crawling_and_indexing_algorithm_for_the_application_of_smart_web_contents

### 2. A Novel Multi-Threaded Web Crawling Model (Weijie Jiang, 2024)

This paper presents an **optimized queue-based web crawling system** using **multi-threading**. The authors propose a **buffer queue system** where multiple threads:

1. Fetch web pages in parallel.
2. Store them in a shared buffer queue.
3. **Use priority-based ranking** to determine which links to crawl next. The study demonstrates that **multithreading increases efficiency by up to 60% compared to single-threaded crawlers**, reducing bottlenecks in web scraping.

**Link to the research paper** - https://arxiv.org/html/2407.10440v1

### 3. On Multi-Thread Crawler Optimization for Scalable Text Searching (Guang Sun, 2019)

This research compares **BFS (Breadth-First Search) and DFS (Depth-First Search)** crawling strategies in a **multi-threaded environment**. The study finds that **DFS-based crawlers** work better for small, focused searches, while **BFS-based crawlers** are more efficient for large-scale crawling. Additionally, the paper discusses **thread synchronization techniques**, ensuring that threads do not fetch duplicate URLs.

**Link to the research paper** - https://www.techscience.com/jbd/v1n2/29019

### 4. Designing Web Crawler Based on Multi-Threaded Approach for Authentication of Web Links on Internet (Kuldeep Vayadande, 2022)

This paper explores a **priority-based web crawler** that **authenticates links before crawling**. The authors propose:

1. Using **priority queues** to schedule crawling tasks efficiently.
2. **Thread-safe crawling** mechanisms to prevent data corruption.
3. **Verification techniques to avoid duplicate or broken links**. This study contributes valuable insights into **thread management in web crawling**, making it relevant to our project.

**Link to the research paper** - https://ieeexplore.ieee.org/abstract/document/10009614

---

# Proposed Approach

## 1. AI-Based Smart URL Prioritization

We will implement **Reinforcement Learning (RL) or Neural Networks** to prioritize URLs dynamically based on relevance and crawling success rate.

```
import numpy as np
from sklearn.ensemble import GradientBoostingClassifier

# Sample training data: [URL Length, Number of Outbound Links, Keyword Score]
```

```python
X_train = np.array([[50, 10, 0.8], [70, 20, 0.9], [30, 5, 0.6]])
y_train = np.array([1, 1, 0])  # 1 = High Priority, 0 = Low Priority

model = GradientBoostingClassifier()
model.fit(X_train, y_train)

# Predict priority of a new URL
def predict_priority(url_features):
    return model.predict([url_features])[0]
```

---

## 2. Multi-Threading Implementation Using pthreads

We will create multiple threads where each **fetches a web page concurrently**.

```c
#include <pthread.h>
#include <curl/curl.h>
#include <stdio.h>

void *crawl_page(void *url) {
    CURL *curl = curl_easy_init();
    if (curl) {
        curl_easy_setopt(curl, CURLOPT_URL, (char *)url);
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
        curl_easy_perform(curl);
        curl_easy_cleanup(curl);
        printf("Crawled: %s\n", (char *)url);
    }
    return NULL;
}
```

---

## 3. Incremental Crawling to Avoid Recrawling

We will store the last modified timestamp for each page and only crawl new or updated content.

```python
import os
import hashlib

def has_changed(url, content):
    hash_val = hashlib.md5(content.encode()).hexdigest()
    if os.path.exists(f"cache/{url}.hash"):
        with open(f"cache/{url}.hash", "r") as f:
            old_hash = f.read().strip()
        if old_hash == hash_val:
```

```python
        return False
    with open(f"cache/{url}.hash", "w") as f:
        f.write(hash_val)
    return True
```

---

### 4. Storing and Indexing Crawled Data Efficiently

We store web pages in **SQLite for fast retrieval and optimize storage using compression**.

```c
#include <sqlite3.h>
#include <zlib.h>

sqlite3 *db;
void init_db() {
    sqlite3_open("crawler.db", &db);
    sqlite3_exec(db, "CREATE TABLE IF NOT EXISTS pages (url TEXT, content BLOB);", 0,
0, 0);
}

void save_page_to_db(const char *url, const char *content) {
    char compressed_content[1024];
    uLong compressed_size = sizeof(compressed_content);
    compress((Bytef *)compressed_content, &compressed_size, (const Bytef *)content,
strlen(content));

    sqlite3_stmt *stmt;
    sqlite3_prepare_v2(db, "INSERT INTO pages (url, content) VALUES (?, ?);", -1, &stmt, 0);
    sqlite3_bind_text(stmt, 1, url, -1, SQLITE_STATIC);
    sqlite3_bind_blob(stmt, 2, compressed_content, compressed_size, SQLITE_STATIC);
    sqlite3_step(stmt);
    sqlite3_finalize(stmt);
}
```

---

# Final Implementation Plan

- Use AI to prioritize high-value URLs.
- Multi-threaded crawling with pthreads.
- Incremental crawling to avoid redundant recrawling.
- Optimized storage using SQLite & compression.

This approach ensures that our **multi-threaded web crawler efficiently fetches, stores, and indexes web pages while using AI to enhance prioritization and reduce redundant processing**, making it highly scalable and intelligent.